# INTLAB - The Matlab/Octave toolbox for Reliable Computing

Siegfried M. Rump[1,2]

[1]Institute for Reliable Computing, Hamburg University of Technology, Am Schwarzenberg-Campus 3, Hamburg, 21073, Germany.
[2]Faculty of Science and Engineering, Waseda University, 3–4–1 Okubo, Shinjuku-ku, Tokyo, 169–8555, Japan.

**Abstract**

The Matlab/Octave toolbox INTLAB offers many so-called verification algorithms the results of which are correct with mathematical certainty. Starting 1998 INTLAB is continuously enhanced. The latest Version 14.1 comprises of 2,083 m-files with more than 76,000 lines of code (with comments more than 137,000). There are several thousand users in more than 50 countries. We describe fundamentals and latest developments in INTLAB including fast and accurate real and complex, full and sparse matrix residuals, sparse linear, in particular nearly symmetric as well as nonlinear systems, and matrix decompositions. Several executable Matlab/Octave codes are included throughout the sections.

**Keywords:** INTLAB, reliable computing, numerical analysis, full and sparse linear and nonlinear systems, optimization, mathematically rigorous error bounds

## 1 Introduction

INTLAB [19] is entirely written in Matlab [17] and works solely on the set of double precision floating-point numbers $\mathbb{F}$. Nevertheless, all computed results including input and output are correct with mathematical certainty. For an overview of verification methods see [18, 22]. Following we give an introduction into the principles of INTLAB, starting with the arithmetic and input and output. Each following section introduces foundations and examples for the solution of standard problems in numerical analysis. We give several applications where input starts with `>>` in black and output in blue, using a laptop with Intel® i7-8550U CPU with 1.8 GHz and 16 GB RAM.

There are 25 demo files in [9] giving more detailed information to each section.

## 2 Input/output, directed rounding and arithmetic

INTLAB uses intervals to represent real/complex data which are not representable in floating-point. A typical definition of an interval as infimum and supremum is

```
>> format short infsup, C = infsup(2.5,4)
intval C =
[    2.5000,    4.0000]
```

The interpretation is that `C` comprises of *all real* numbers $\{x \in \mathbb{R} : 2.5 \leq x \leq 4\}$. Defining an interval by midpoint and radius may be more convenient for small radii:

```
>> P = midrad(pi,1e-12)
intval P =
[    3.1415,    3.1416]
```

All displayed output is correct in the mathematical sense, i.e., `P` is included in `[3.1415,3.1416]`. Note that this is the narrowest inclusion in the given format "short", it does not reflect the accuracy of the inclusion. The latter can be seen by

```
>> format long, P
intval P =
[   3.14159265358879,   3.14159265359080]
```

or better in mathematically correct mid-rad notation by

```
>> midrad(P)
intval P =
<   3.14159265358979,  0.00000000000101>
```

A convenient display for not too wide intervals is the "_"-notation:

```
>> format _, P
intval P =
   3.14159265359___
```

In that case subtract and add 1 from the last displayed figure to obtain a correct inclusion interval, i.e., $[3.14159265358, 3.14159265360]$ is a correct inclusion of `P`. Sunaga [30] used a similar notation in his Master Thesis at the University of Tokyo in 1956.

Care is necessary when real numbers not being a floating-point number are used as a bound. For example, the command `X = intval(0.8)` first converts "0.8" into $\mathbb{F}$ and forms then an interval with coinciding left and right bound. Similarly, for

```
>> X = infsup(0.8,2); getbits(X)
ans =
  4x63 char array
    'infimum                                                       '
    ' +1.10011001100110011001100110011001100110011001100110011010 * 2^-1'
    'supremum                                                      '
    ' +1.00000000000000000000000000000000000000000000000000000000 * 2^1 '
```

it is clear from the bit pattern that the lower bound of `X` is larger than 0.8. In order to include the real number "0.8" (which is not in $\mathbb{F}$) use `intval('0.8')`, or similarly `intval('pi')` for the narrowest inclusion of the transcendental number $\pi$ in $\mathbb{F}$.

Interval arithmetic in INTLAB is implemented using directed rounding as defined in IEEE 754 [8]. The command `setround(-1)` switches the rounding to downwards

until the next call of `setround`. That implies that for all subsequent operations the computed result is less than or equal to the real result including vector and matrix operations. For example, let `H` be the $3 \times 3$-Hilbert matrix scaled to integers by $\mathtt{H}_{ij} := 60/(i+j-1)$, and let `R` be a preconditioner. Then `M = R*A` in rounding downwards satisfies $\mathtt{M}_{ij} \leq (RA)_{ij}$ for $1 \leq i, j \leq 3$, and similarly for rounding upwards. Therefore $\mathtt{Resinf}_{ij} \leq (RA-I)_{ij} \leq \mathtt{Ressup}_{ij}$:

```
>> format short, n = 3; A = hilbert(n); R = inv(A);
   setround(-1), Resinf = R*A - eye(n);
   setround(+1), Ressup = R*A - eye(n);
   Diam = Ressup - Resinf

Diam =
   1.0e-13 *
    0.0533    0.0356    0.0267
    0.2843    0.2843    0.2132
    0.2132    0.1421    0.1421
```

Hence [`Resinf`,`Ressup`] is a narrow inclusion of $RA - I$. Another way is to use

```
>> Res = R*intval(A) - eye(n)

intval Res =
  1.0e-013 *
[   0.0710,   0.1244] [   0.0088,   0.0445] [   0.0177,   0.0445]
[   0.0000,   0.2843] [   0.0710,   0.3553] [  -0.0711,   0.1422]
[   0.0000,   0.2132] [  -0.0711,   0.0711] [  -0.0711,   0.0711]
```

Now `Res` is, as indicated, an interval matrix including the residual $RA-I$. The rule is that if at least one operand in an operation is of type `intval`, then the corresponding interval operation is used. We made considerable effort to make interval operations fast. One measure is mid-rad representation for the multiplication of interval matrices allowing to use Level-3 BLAS [23]. For comparison, INTLAB allows to use inf-sup representation for multiplication as well with quite some price in computing time:

```
>> n = 5000; A = midrad(randn(n),1e-12); B = midrad(randn(n),1e-12);
   intvalinit('SharpIVmult',0), tic, Csharp = A*B; tsharp = toc
   intvalinit('FastIVmult',0), tic, Cfast = A*B; tfast = toc

tsharp =
   12.8278
tfast =
    5.7582
```

As can be seen the default mid-rad is twice as fast as inf-sup, whereas accidentally

```
>> errSharp = median(relerr(Csharp(:)))
   errFast = median(relerr(Cfast(:)))

errSharp =
   1.6747e-10
errFast =
   1.6738e-10
```

mid-rad is in the median more accurate. That is not unusual for narrow intervals [23].

More details are in [9, Demo 1, 2, 3, 17, 21, 22 and 24].

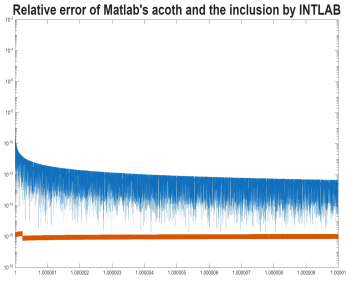
# 3 Complex arithmetic and standard functions

INTLAB provides all elementary standard functions for real and complex, point and interval arguments [20], and in addition some higher transcendental functions including erf, erfc, gamma or gammaln. The inclusions mostly differ by few bits, and under any circumstances they are correct. For example, `sin(infsup(1,2))` must include $\{\sin(x) : 1 \le x \le 2\} = [\min(\sin(1), \sin(2)), 1]$:

```
>> format long, V = sin(infsup(1,2))

intval V =
[   0.84147098480789,   1.00000000000000]
```

Some of the built-in Matlab standard functions have poor quality, for example the inverse hyperbolic cotangent near 1:

```
>> x = linspace(1+1e-14,1+1e-5,100000);
   y = acoth(x); Y = acoth(intval(x));
   semilogy( x,relerr(y,Y), x,relerr(Y) )
```




The relative error of the inclusions by INTLAB are printed in red, the relative error of the built-in Matlab standard function in blue. As can be seen, for arguments close to 1 only some 10 figures of the Matlab approximation are correct.

Another example is the gamma function for negative arguments:

```
>> x1 = -171.25; approx1 = gamma(x1), Gamma_x1 = gamma(intval(x1))

approx1 =
   Inf
intval Gamma_x1 =
  1.0e-309 *
   0.98910291950447
```

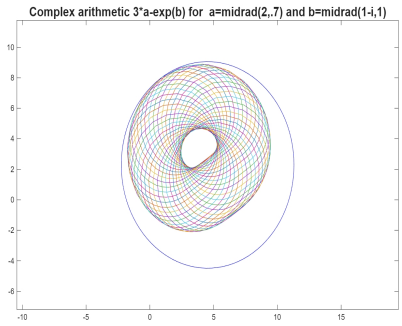
Obviously Matlab cannot handle large negative arguments. But worse,

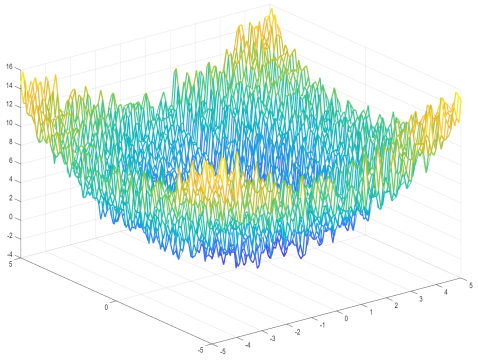
```
>> x2 = -172.25; approx2 = gamma(x2), Gamma_x2 = gamma(intval(x2))

approx2 =
   Inf
intval Gamma_x2 =
  1.0e-311 *
  -0.5742252072588_
```

the approximation of $\Gamma(x_2)$ is still $+\infty$ but must mathematically have opposite sign of $\Gamma(x_1)$. The inclusion by INTLAB is narrow guaranteeing almost all digits.

For interval arithmetic over complex numbers it is appropriate to use mid-rad arithmetic [7, 30]. This naturally imposes some overestimation. For example, `3*a - exp(b)` for `a=midrad(2,.7), b=midrad(1-i,1)` yields the left picture below:

```
>> close, kmax = 40; a = midrad(2,.7); b = midrad(1-1i,1);
plotintval(3*a-exp(b)); hold on
phi = linspace(0,2*pi,kmax);
[A,B] = meshgrid(mid(a)+rad(a)*exp(i*phi),mid(b)+rad(b)*exp(i*phi));
plot(3*A-exp(B))
```



Programs in INTLAB are vectorized where possible with quite some gain in computing time. We compare the computation of an inclusion of $e^x$ for some $n = 100,000$ mesh points between 1 and 10, first with a loop and then vectorized.

```
>> n = 1e5; x = linspace(1,10,n); format short
   tic, y = intval(x);
   for k=1:n, y(k) = exp(intval(x(k))); end
   tloop = toc
   tic, Y = exp(intval(x)); Tvectorized = toc
tloop =
  25.9993
Tvectorized =
   0.0067
```

Due to the interpretation overhead in Matlab it is a general rule that vectorized code should be used and loops be avoided. All routines in INTLAB follow that paradigm.

The function evaluation of $f$ for interval arguments $X, Y$ guarantees that the result $Z = f(X, Y)$ satisfies $\{f(x, y) : x \in X, y \in Y\} \subseteq Z$. That makes interval methods in particular attractive to optimization problems. Consider Trefethen's function

$$f(x,y) = e^{\sin(50x)} + \sin(60e^y) + \sin(70\sin(x)) + \sin(\sin(80y)) - \sin(10(x+y)) + (x^2+y^2)/4$$

who asked for the global minimum in problem 4 the SIAM $10 \times 10$-digit challenge [32], see the plot on the right for $[-5, 5]^2$. When evaluating the function on $X := [-10, 10]^2$ with 10,000 mesh points in each direction, Matlab needs 6.8 seconds to approximate the minimum on $X$ to be $-3.307$, whereas INTLAB needs 0.009 seconds to verify that $\{f(x) : x \in X\} \subseteq [-3.474, 19.06]$. Only the lower bound is important. Matlab's bound is better, but not guaranteed, the true minimum might have been missed (see Section 9 for an extreme example), and increasing the number of mesh points cannot cure that principle problem. We come to Trefethen's example again in Section 9.

More details and literature on this section are in [9, Demo 1, 3 and 9].

# 4 Fast and accurate matrix operations

Accurate matrix computations and residuals are crucial for verification algorithms. With Version 14 of INTLAB the routines "`prodK`" and "`spProdK`" are introduced. Both are written by Marko Lange, and the background will be published in [13]. The algorithms compute dot product expressions such as vector or matrix residuals in a very versatile manner, with high accuracy, as approximation and/or inclusion as well as results represented as unevaluated sums. Various measures speed up the algorithms such as Karatsuba's method and a nearly optimal choice of splitting points, cf. [13].

Consider a linear system with the $12 \times 12$ inverse Hilbert matrix $H$ and first column of the identity matrix as right hand side $b$, so that $(A^{-1}b)_i = 1/i$. Matlab's initial approximation is poor but improved by a residual iteration:

```
>> n = 12; A = invhilb(n); b = zeros(n,1); b(1) = 1;
   xs = A\b; x = 1./(1:n)'; r = [];
   for k=1:10
     if k~=1, xs = xs - A\prodK(A,xs,-1,b,2); end
     r = [r max(relerr(xs,x))];
     if mod(k,5)==0, disp(r), r = []; end
   end
  1.8096e-02   5.5165e-04   1.7118e-05   5.3147e-07   1.6501e-08
  5.1230e-10   1.5905e-11   4.9377e-13   1.5321e-14   2.2204e-16
```

As expected, the maximum error of the approximation decreases according to the condition number. Next we compute inclusions of the matrix residual $RA - I$ for a matrix $A$ of dimension $n$ and a preconditioner $R$, generated by the first line below.

```
>> A = randn(n); R = inv(A);
   [res,delta] = prodK(R,A,-1,eye(n),3);
   mp.Digits(34);
   setround(-1), resinf = double(R*mp(A)-eye(n));
   setround(+1), ressup = double(R*mp(A)-eye(n));
 n = 1000, t_prodK =  0.26, t_mp =   12.9, ratio =  49.5
 n = 2000, t_prodK =  1.20, t_mp =  113.0, ratio =  94.1
 n = 5000, t_prodK = 15.64, t_mp = 3753.8, ratio = 240.0
```

The second line computes matrices `res, delta` using INTLAB's `prodK` such that `res` $-$ `delta` $\leq RA-I \leq$ `res` $+$ `delta` with entrywise comparison. The last parameter $k = 3$ indicates calculation in $(k/2+1)$- or 2.5-fold precision. Alternatively, the mp-toolbox Advanpix [6] can be used. With the command `mp.Digits(34)` a particularly fast implementation of extended precision is used fully satisfying the IEEE 754 arithmetic standard and respecting the rounding mode. There is no interval arithmetic, so we calculate $RA - I$ in Lines 4 and 5 in rounding downwards and upwards.

All results are close to maximally accurate, but `prodK` is some 2 orders of magnitude faster than the mp-toolbox [6], the latter written in highly-optimized C-code. Due to the author Pavel Holoborodko of [6] the reason is the significant overhead by Matlab for custom data types and a time-consuming process of creating new mp-objects.

For sparse matrices there is less data transfer for the mp-toolbox. Following we generate matrices $A$ with 0.02% nonzero elements per row and use INTLAB's routine `spProdK` for fast and accurate matrix multiplication.

```
>> A = sprandn(n,n,0.0002);
   tic, spProdK(A,A,2); tspProdK = toc;
   tic, double(A*mp(A)); tmp(k) = toc;
n =  10,000, t_spProdK =  0.02, t_mp =  0.04, ratio = 2.0
n =  20,000, t_spProdK =  0.06, t_mp =  0.18, ratio = 3.0
n =  50,000, t_spProdK =  0.64, t_mp =  1.12, ratio = 1.7
n = 100,000, t_spProdK =  6.64, t_mp =  5.73, ratio = 0.9
```

Still Marko Lange's Matlab routine `spProdK` is faster than `mp`.

It is shown in [21] that extremely ill-conditioned matrices can be inverted in double precision if accurate dot products are available. Following `A = randmat(n,cnd)`

```
>> n = 100; cnd = 1e25;
   A = randmat(n,cnd); R = inv(A); X = prodK(R,A,3);
   RR = prodK(inv(X),R,3,'OutputTerms',2);
   Res = prodK(RR,A,-1,eye(n),3);
   normRes = norm(Res), condA = norm(RR{1})*norm(A)
normRes =
   1.4977e-05
condA =
   6.0042e+26
```

generates an $n \times n$-matrix with anticipated condition number `cnd`. We choose `cnd` $= 10^{25}$ which is way above the maximum condition number $10^{16}$ what double precision can handle. As a consequence, the "approximate" inverse `R` differs by more than 5 orders of magnitude from $A^{-1}$, and half of the entries have wrong sign.

Nevertheless we calculate the product `X = R*A` in 2.5-fold precision and store it in double precision. The floating-point inverse of `X` is multiplied by `R`, so that without rounding errors $(RA)^{-1}R = A^{-1}$. The product is stored in an unevaluated sum `RR{1}+RR{2}`, and $\|\texttt{RR} * \texttt{A} - I\| \approx 10^{-5}$ shows that `RR` is indeed a reasonable preconditioner. Note that the condition number of `A` exceeds $10^{26}$.

In [21] that principle is iterated by storing the approximate inverse as an unevaluated sum of more terms. The code of Algorithm `invIllco` in INTLAB

```
>> k = 1;
   while 1
     k = k+1; P = prodK(R,A,2*k-1); X = inv_(P);
     R = prodK(X,R,2*k-1,'OutputTerms',k);
     N = norm(prodK(R,A,-1,eye(n),k+5),inf);
     if N<1e-15, break, end
   end
```

is basically as above. For extremely ill-conditioned matrices sometimes Matlab's "`inv`" fails to invert `P`. To that end we wrote "`inv_`" which perturbs the input matrix `P` slightly in case of failure. In [21] we show an example of a matrix with condition number $10^{305}$. The code above computes $A^{-1}$ stored as an unevaluated sum `R{:}` of 22 terms with maximum accuracy, solely using double precision and the accurate matrix residuals by `prodK`.

More examples for `prodK` and `spProdK` as well as much more details are in [9, Demo 3, 5, 6 and 25].

# 5 Dense systems of linear equations

The solution of linear systems is ubiquitous in numerical analysis. There are two aspects, timing and accuracy. Comparison with Matlab's backslash operator is comparing apples with oranges: backslash delivers an approximation without accuracy information (see below), whereas INTLAB's verification routines' results are correct with mathematical certainty. Having said this, we take a look at timing[1]

```
>> for n=[1000 2000 5000 10000]
     A = randn(n); b = randn(n,1);
     tic, A\b; tapprox = toc;  tic, A\intval(b); tincl = toc;
   end
n =  1000, tapprox =  0.01, tincl =   0.22, ratio =  20.3
n =  2000, tapprox =  0.08, tincl =   1.10, ratio =  13.1
n =  5000, tapprox =  0.81, tincl =  13.45, ratio =  16.6
n = 10000, tapprox =  5.42, tincl = 100.13, ratio =  18.5
```

As can be seen there is a price to pay for safety. However, for sparse systems verification is often faster and more accurate than Matlab's backslash, see the next section.

Next an ill-conditioned example, the inverse Hilbert matrix $H$ for $n = 12$. Following we solve $Hx = e_1$ for $e_1$ denoting the first column of the identity matrix as in Section 4 and list the first 4 entries of the approximation $A\backslash b$ and the inclusion $A\backslash\text{intval}(b)$.

```
        A\b                 A\intval(b)
 0.99340260526271      1.0000000000____
 0.49407300009058      0.500000000000__                    (4.1)
 0.32795052064139      0.3333333333333_
 0.24506826963778      0.25000000000000
```

The true solution is the first column of the original Hilbert matrix, i.e., $1/i$ for $1 \le i \le n$. Matlab's approximation by backslash has just 1 correct figure, the inclusion is accurate to at least 11 figures. In this example Matlab produces a warning that the linear system is ill-conditioned. In fact, the condition number exceeds $10^{16} \approx \mathbf{u}^{-1}$.

The next example was constructed by Wright [33] to demonstrate an exponential growth factor. The boundary value problem $\dot{x} = x - 1$ with $x(0) = x(40)$ on $[0, 40]$ is discretized using the trapezoidal rule. The solution is $x \equiv 1$, and we choose $n = 64$.

```
        A\b                 A\intval(b)
   1.00000000000000     1.00000000000000
   1.00000000000000     1.00000000000000

 -12.04231844530580     1.0000000000000_
  12.19089533191109     1.00000000000000
   3.90551943731236     1.0000000000000_
   1.00000000000000     1.00000000000000
```

We display the first 2 and last 4 entries of the solution (which should be identically 1). The last entries of the approximation by backslash are grossly wrong, sometimes with incorrect sign, where INTLAB's inclusion guarantees at least 15 correct digits of all entries. Matlab does not perform one residual step in working precision as this would imply backward stability [29]. Since the matrix has condition number less than

[1] We display executable code but henceforth omit sometimes lines as printing the result or alike.

30, we expect at least 14 correct digits of an approximate solution. However, Matlab's approximation is grossly wrong and comes without warning.

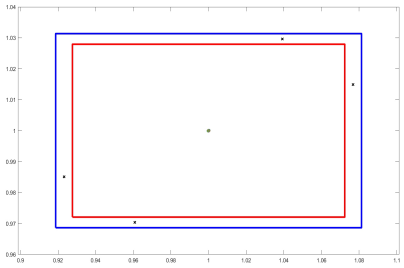
The next example is extremely ill-conditioned. We scale the Hilbert matrix by the least common multiple of $1, \ldots, 2n-1$ to produce integer entries and want to compute the last column of the inverse for $n = 18$. The call `verifylss(A,b)` does not succeed,

| (A\b)_{1..5} | X_{1..5} | (A\b)_{14..18} | X_{14..18} |
|---|---|---|---|
| -0.00000031 | -5.7____e-004 | -79.48295002 | 1.6122_e+008 |
| 0.00005183 | 0.1731_ | 46.88417066 | -1.0199_e+008 |
| -0.00213765 | -13.1538_ | -14.35304864 | 4.3517_e+007 |
| 0.03866674 | 438.462__ | 1.15791450 | -1.1219_e+007 |
| -0.38334781 | -8.0567_e+003 | 0.29924026 | 1.3199_e+006 |

but there is an option for extremely ill-conditioned matrices following [21]. We display left the first and right the last 5 entries of Matlab's approximation and INTLAB's inclusion `X = verifylss(A,b,'illco')`. The approximation is off by orders of magnitudes, however, in this case a warning is given.

INTLAB's linear system solver `verifylss` accepts interval linear systems, i.e., the matrix $\mathbf{A}$ and right hand $\mathbf{b}$ varying within given tolerances. It is NP-hard to compute a sharp inclusion of the solution set $\Sigma(\mathbf{A}, \mathbf{b}) := \{x \in \mathbb{R}^n : \exists A \in \mathbf{A}\ \exists b \in \mathbf{b} \text{ with } Ax = b\}$.

```
>> n = 1000;
   A = midrad(randn(n),1e-6); b = A.mid*ones(n,1); cnd = cond(A.mid)
   tic, [X,Xinf,Xsup] = verifylss(A,b); t_verifylss = toc
   Xinner = infsup(Xinf,Xsup);
   plotintval(X(1:2),'n'), hold on, plotintval(Xinner(1:2),'n')
   tic
   for i=1:5000
     AMC = A.mid + ( 2*(rand(size(A))>.5)-1 ).*rad(A);
     x = AMC\b; plot(x(1),x(2),'o')
   end
   t_MonteCarlo_5000 = toc
cnd =
   1.6179e+03
t_verifylss =
    0.2147
t_MonteCarlo_1000 =
  126.2222
```



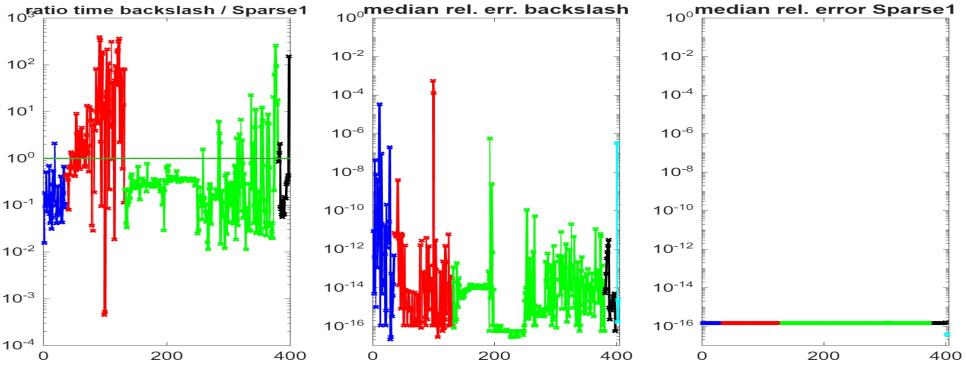
We generate a random $1000 \times 1000$ matrix $A$ with condition number $1.6 \cdot 10^3$, entrywise radii $10^{-6}$ and use $b := A \cdot (1, \ldots, 1)^T$. The output `X` of `verifylss` is an outer inclusion of $\Sigma(\mathbf{A}, \mathbf{b})$, i.e., it satisfies $\Sigma(\mathbf{A}, \mathbf{b}) \in$ `X`. The extra parameters `Xinf, Xsup` are inner inclusions of $\Sigma(\mathbf{A}, \mathbf{b})$, i.e., for $1 \le i \le 1000$ there exist $A_1, A_2 \in \mathbf{A}$ with $(A_1^{-1}b)_i \le$ `Xinf`$_i$ and `Xsup`$_i \le (A_2^{-1}b)_i$. A Monte Carlo approach solves $Ax = b$ for 5,000 matrices $A$ on the boundary of $\mathbf{A}$, and the first and second entry of each solution is plotted as $\circ$ in the graph above. In addition we use sign information of $A^{-1}$ to solve 4 particular linear systems and plot the solutions as $\times$. The Monte Carlo solutions in the middle are far from the boundary of $\Sigma(\mathbf{A}, \mathbf{b})$, whereas the inner inclusions in red prove that `X` is close to optimal. More examples and details are in [9, Demo 1, 3, 4 and 25].

# 6 Sparse systems of linear equations

Solving sparse systems of linear equations was a long standing open problem in verification methods [22, Challenge 10.15]. In Part I and Part II of [28] we present two methods which are included in INTLAB Version 14 as Algorithms "`verifySparselss1/2`". They work for large dimension and condition number. They are mostly more accurate than Matlab's backslash operator and often even faster. In [28] we tested the algorithms for all 403 real and complex examples in the Florida collection [5] with

$$10^3 \le n \le 10^5, \quad 10^7 \le \kappa_2(A) \le 10^{16} \quad \text{and up to 1 million nonzero entries .}$$

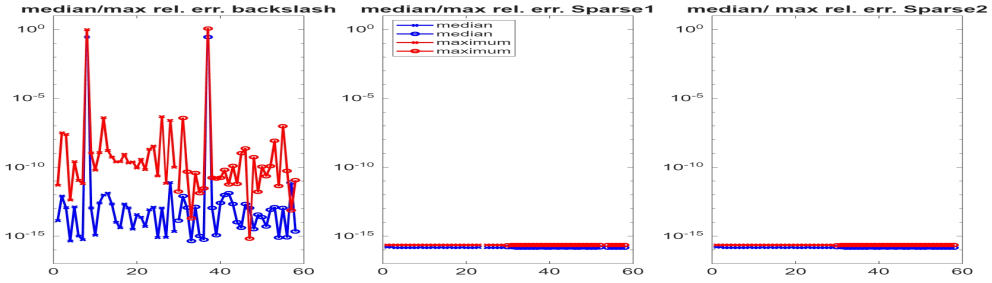
The following pictures are taken from [28]. Left is the time ratio of the backslash




operator versus our verification routine which is 0.2 in the median over all examples. However, all numbers above 1 mean that INTLAB's verification is faster than Matlab's backslash. In the middle we see the median relative error of backslash over all entries. So in the median some 14 figures are correct, but often less. On the right is the median relative error of INTLAB's verified result. As can be seen in the median the inclusion is maximally accurate for all entries. Moreover, we tested all rectangular systems with

$$10^3 \le \max(m,n) \le 10^5 \quad \text{and} \quad 10^7 \le \kappa_2(A) \le 10^{16} \quad \text{and} \quad \texttt{nnz(A)} \le 10^6 \ ,$$

in total 2 least squares and 27 underdetermined problems. In order to increase the number of tests we tested the original samples and in addition $A^T$ leading to 2 underdetermined and 27 least squares problems. The left graph shows the median (blue)




and maximum (red) relative error of Matlab's backslash or "`lsqminnorm`", the middle and right the median and maximum relative error of our verification Algorithms. INTLAB's inclusions are always maximally accurate.

The Matlab backslash operator selects automatically among different numerical algorithms depending on the matrix properties, such as sparsity, symmetry, and conditioning. In practical examples, symmetry is sometimes jeopardized by rounding errors, i.e., the matrix $A$ is "nearly symmetric" in the sense that $A^T \neq A$ but $\|A^T - A\|_\infty \leq \varphi \|A\|_\infty$ for very small $\varphi$. In such a case Matlab is forced to ignore all algorithms for symmetric input matrix.

Both sparse solvers `verifySparselss1/2` are based on and choose automatically between individual subalgorithms for spd, symmetric and general matrices. If all diagonal entries of symmetric/Hermitian $A$ have the same sign, the spd solver is tried. If successful, positive definiteness was proved a posteriori, otherwise the symmetric solver is tried. If it fails as well, `verifySparselss1/2` calls the respective general solver.

All subalgorithms are based on a lower bound of the smallest singular value of $A$ or of the augmented matrix $\begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix}$ so that

$$\|A^{-1}b - \tilde{x}\|_\infty \leq \|A^{-1}b - \tilde{x}\|_2 \leq \sigma_{\min}(A)^{-1} \|b - A\tilde{x}\|_2.$$

A nearly symmetric matrix with $S \approx (A^T + A)/2, \Delta := S - A$ and $\|\Delta\|_2 \leq \alpha$ satisfies

$$\|A^{-1}b - \tilde{x}\|_\infty \leq \|(S - \Delta)^{-1}(b - A\tilde{x})\|_2 \leq (\sigma_{\min}(S) - \alpha)^{-1} \|b - A\tilde{x}\|_2$$

so that using the spd or symmetric subalgorithm for a lower bound on $s := \sigma_{\min}(S)$ and provided that $s > \alpha$ gives verified bounds for $\|A^{-1}b - \tilde{x}\|_\infty$. This is the principle, in reality Algorithms `verifySparselss1/2` for nearly symmetric matrices are much more involved, cf. [28]. The sparse solvers try the nearly symmetric case if the defect from symmetry $\delta := \|A^T - A\|_\infty / \|A\|_\infty$ is small, or can be forced to do that.

**Table 1** Nearly symmetric input matrix [the inclusions for all entries are maximally accurate]

| # | n | nnz | $\kappa_\infty$ | $\delta$ | $t_\backslash$ | $t_{gen}$ | $t_{sym}$ | rel. error backslash median | rel. error backslash max |
|---|---|---|---|---|---|---|---|---|---|
| 801 | 48,600 | 1,181,120 | 1.7e5 | 8.8e-17 | 8.9 | 90.4 | 5.0 | 8.4e-15 | 1.5e-10 |
| 802 | 157,464 | 3,866,688 | 1.4e5 | 8.8e-17 | 103.4 | 832.7 | 20.0 | 8.1e-15 | 2.0e-9 |
| 930 | 12,504 | 874,887 | 1.7e8 | 6.8e-17 | 27.1 | 30.1 | 1.8 | 1.9e-12 | 4.1e-8 |
| 931 | 29,067 | 2,081,063 | 1.1e8 | 9.1e-17 | 1000.2 | 194.9 | 4.3 | 1.7e-12 | 1.4e-7 |
| 932 | 42,930 | 3,148,656 | 3.2e8 | 9.6e-17 | 3276.8 | 320.3 | 6.4 | 4.3e-13 | 7.6e-8 |
| 1417 | 321,821 | 1,931,828 | 5.1e22 | 7.7e-21 | mem. | 22.0 | 26.9 | | |
| 1428 | 20,082 | 281,150 | 2.2e7 | 3.0e-18 | 1.0 | 30.8 | 1.6 | 4.8e-15 | 9.2e-11 |
| 1851 | 17,880 | 430,740 | 1.7e3 | 4.6e-17 | 2.1 | 23.4 | 1.4 | 2.2e-16 | 9.9e-15 |
| 1852 | 147,900 | 3,489,300 | 7.7e3 | 4.7e-17 | 38.5 | 529.5 | 14.7 | 1.7e-16 | 9.7e-15 |

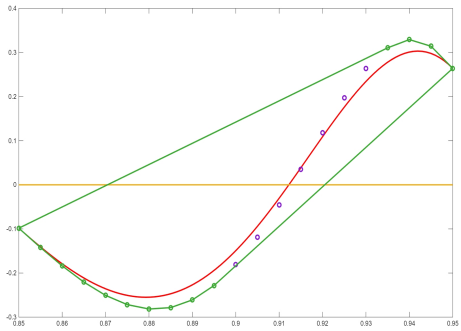
In Table 1 we show for examples of the Florida Sparse Matrix Collection [5] from left the number in [5], dimension, `nnz(A)`, the condition number, the defect $\delta$ followed by the computing times in seconds for Matlab's backslash, INTLAB's `verifySparselss1` without and with using near symmetry. INTLAB's general solver is often slower than Matlab's backslash, as expected, whereas INTLAB's near symmetry is mostly faster, in the two cases 931 and 932 by more than two orders of magnitude. For #1417 in [5] trying near symmetry failed, thus increasing the computing time (cf. [28] for details), where Matlab fails with "out of memory". All inclusions of all entries are maximally accurate allowing to estimate the accuracy of Matlab's backslash operator.

More examples and details are in [9, Demo 3, 4, 5 and 8] are shown.

# 7 Polynomials

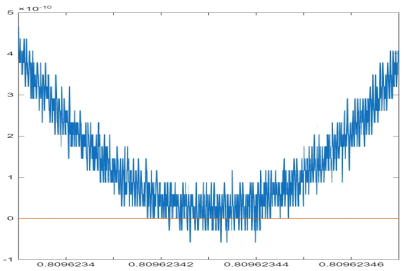
There is a polynomial toolbox in INTLAB for uni- and multivariate, real and complex polynomials with point or interval coefficients. Following we estimate the range of the Legendre polynomial over $[0.85, 0.95]$ using interval arithmetic and Bernstein

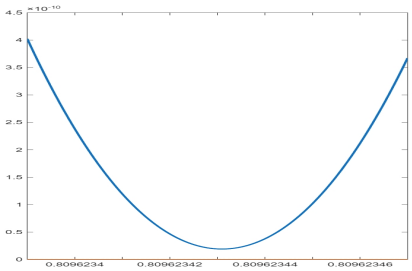
```
>> P = intval(legendre(20)); a = 0.85; b = 0.95;
   X = infsup(a,b);  Yintval = P{X}
   plotpoly(P,a,b), hold on, plotbernstein(bernsteincoeff(P,X),a,b)
   YBernstein = infsup(min(B.c.inf),max(B.c.sup))

intval Yintval =
[ -1.0613e+005,  1.0540e+005]
intval YBernstein =
[   -0.2816,    0.3292]
```



coefficients, the latter following [31]. The command `P{X}` evaluates `P` with interval argument `X`. The result `Yintval` is correct but a huge overestimation of the true range $R = \{P(x) : x \in X\} = [-0.26, 0.31]$. The circles in the graph on the right show the Bernstein coefficients, the convex hull of which includes the range. The resulting inclusion `YBernstein = [-0.29,0.33]` is close to the true range $R$.

Consider the polynomial $Q = 194481x^4 - 629748x^3 + 993178x^2 - 782664x + 233290$. We ask whether $Q(x)$ has a real root near $\tilde{x} = 0.80962343$. To that end we plot the polynomial in $X = \tilde{x} \pm 4 \cdot 10^{-8}$, see the left graph. The problem is too ill-conditioned for evaluation in double precision, it cannot be decided whether there is a root of $Q$





or not. We developed a pair arithmetic [11] interpreting a pair $(a, \delta)$ as unevaluated sum $a + \delta$ with corresponding arithmetic operations. It is simpler than double-double arithmetic [3] and satisfies that the midpoint of an operation is the operation on the midpoints. Both pair arithmetic are included in INTLAB as "`cpair`" and "`dd`". Evaluating $Q$ over $X$ using `cpair` decovers that there is no real root in $X$.

There is a long arithmetic "`long`" with point and interval operations in INTLAB as well, where a higher computing precision and exponent range can be specified, see also Section 13.

More examples and details on this section are in [9, Demo 1, 13 and 20].

# 8 Gradient, Hessians, slopes, Taylor series

Algorithmic differentiation is a fundamental tool for fast and accurate verification routines. INTLAB offers the `gradient, Hessian, slope and Taylor` toolbox. Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be sufficiently smooth with a Matlab/Octave function `f` for computing $f(x)$. Then `y = f(gradientinit(xs))` produces approximations `y.x` and `y.dx` for $f(\tilde{x})$ and the derivative/Jacobian at $\tilde{x}$, respectively. Similarly, `y = f(hessianinit(xs))` produces in addition an approximation `y.hx` for the second derivative/Hessian at $\tilde{x}$.

Following we use the interval Newton method [22] to calculate an inclusion of $\sqrt{5}$, the root of $f(x) = x^2 - 5$. For an interval $\mathbf{X}$ and $\tilde{x} \in \mathbf{X}$ suppose $0 \notin f'(\mathbf{X})$. Define

$$N(\tilde{x}, \mathbf{X}) := \tilde{x} - f(\tilde{x})/f'(\mathbf{X}) \ .$$

If $N(\tilde{x}, \mathbf{X}) \subseteq \mathbf{X}$, then $\mathbf{X}$ contains a unique root of $f$ in $\mathbf{X}$. If $N(\tilde{x}, \mathbf{X}) \cap \mathbf{X} = \emptyset$, then $f(x) \neq 0$ for all $x \in \mathbf{X}$. This holds for multivariate functions as well, cf. [19, Theorem 13.1]. The "_"-display for initial $\mathbf{X} = [2, 4]$ illustrates the quadratic convergence:

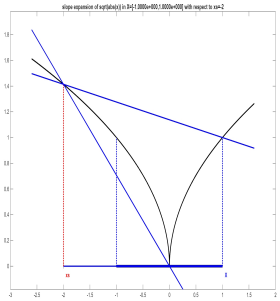
```
>> format _ long,  f = @(x) x^2 - 5;  X = infsup(2,4);
   for k=1:4
     Y = f(gradientinit(X)); X = X.mid - f(X.mid)/Y.dx;
     fprintf('k = %2d,  X = %s\n',k,disp_(X))
   end
k =  1,  X = [   2.00000000000000,   2.50000000000000]
k =  2,  X =    2.24_____________
k =  3,  X =    2.236068_________
k =  4,  X =    2.23606797749979_
```

Next we verify a local minimum of $f(x) := \sum_{i=1}^{n-10} \frac{x_i}{x_{i+10}} + \sum_{i=1}^{n} (x-1)^2 - \sum_{i=1}^{n-1} x_i x_{i+1}$, here for $n = 5,000$. Below the INTLAB function "`verifylocalmin`" applies a Newton iteration using the `gradient` toolbox with starting value $(1, \ldots, 1)^T$ and computes an inclusion `X` $\in \mathbb{IR}^{5000}$ of a stationary point $\hat{x}$. The corresponding nonlinear system uses a sparse solver as in Section 6. The next line computes an inclusion `H` of the Hessians $\{H(x) : x \in$ `X`$\}$ and verifies positive definiteness of all $H \in$ `H`, in particular of $H(\hat{x})$.

```
>> tic, X = verifylocalmin(@f,ones(5000,1)); tmin = toc;
   tic, H = f(hessianinit(X)); isMin = isspd(H.hx); tspd = toc;
   fprintf('t_min %4.2f, t_spd %4.2f, isspd = %d\n',tmin,tspd,isMin)
time min 0.54, time spd 0.08, isspd = 1
```

Hence $\hat{x}$ a local minimum of $f$. The verification for 5,000 unknowns takes totally less than a second.

Another possibility to expand a function are slopes [22]. A slope `S` has three components: An inclusion `S.c` of $f(\tilde{x})$, a superset `S.r` of the range and `S.s` such that $\forall x \in X$ $\exists s \in$ `S.s` : $f(x) = f(\tilde{x}) + s(x - \tilde{x})$. Once initialized by `slopeinit(xs,X)` arithmetic operations and standard functions are available. To the right is a plot of $\sqrt{|x|}$ for $\tilde{x} = -2$ in the interval $X = [-1, 1]$. Note that $\tilde{x}$ is outside $X$.



The nonlinear system solver "`verifynlss`" in Section 12 allows to choose between gradient and slope expansion. Consider Broyden's

function $f(x) = \begin{pmatrix} \sin x_1 x_2/2 - x_2/(4P) - x_1/2 \\ (1-1/(4P))e^{2x_1-e} + ex_2/P - 2ex_1 \end{pmatrix}$ with the uncertain parameter $P := \pi \pm \varrho$ and initial approximation $[0.6; 0.3]$. We compute an inclusion $X \in \mathbb{IR}^2$ such that for all $p \in P$ there exists $\hat{x} \in X$ with $f(\hat{x}) = 0$.

```
>> for rho=[0.04,0.05]
     Xg = verifynlss(@Broyden,[ .6 ; 3 ],'g',[],rho);
     Xs = verifynlss(@Broyden,[ .6 ; 3 ],'s',[],rho);
     intval incl_gradient_slope = [Xg Xs]
   end
intval incl_gradient_slope =
[    0.4764,    0.5236] [    0.4835,    0.5165]
[    3.0874,    3.1947] [    3.0967,    3.1854]
intval incl_gradient_slope =
[       NaN,       NaN] [    0.4752,    0.5248]
[       NaN,       NaN] [    3.0815,    3.2000]
```

For radius $\varrho = 0.04$, i.e., $P = \pi \pm 0.04$ the inclusions using slopes are a little narrower, for radius 0.05 the expansion by gradients could not compute a verified inclusion.

An extension of slopes are Taylor coefficients. The call `y = f(taylorinit(X))` satisfies for real or complex univariate $f$ that `y{k}` $= f^{(k)}(X)/k!$ where $X$ is a real or complex point or interval. For $f(x) = e^{\cos(x-2)-\sin(x)} - \coth(x+1)$ we obtain

```
>> f = @(x) exp(cos(x-2)-sin(x)) - coth(x+1);
   x = 3.5; e = infsup(0,1);
   y_intval = f(x+e)
   y = f(slopeinit(x,x+e)); y_slope = y.r
   y = f(taylorinit(x));
   y_Taylor = sum( y{0:4} .* e.^(0:4) )
   y_range = verifyrange(f,x+e)
intval y_intval =
[   -0.3629,    1.8527]
intval y_slope =
[   -0.3080,    1.1372]
intval y_Taylor =
[    0.1125,    0.6184]
intval y_range =
[    0.1928,    0.5243]
```

where `y_intval` is the interval evaluation of $f(X)$ for $X = [3.5, 4.5]$, `y_slope` is the range by slopes, `y_Taylor` the inclusion by Taylor arithmetic, and `y_range` is a very narrow inclusion of the true range $[0.1928, 0.5241]$ of $f(X)$ computed in INTLAB using optimization techniques, see Sections 9 and 11. The range inclusion by Taylor arithmetic can verify that $f(x) \neq 0$ for all $x \in X$.

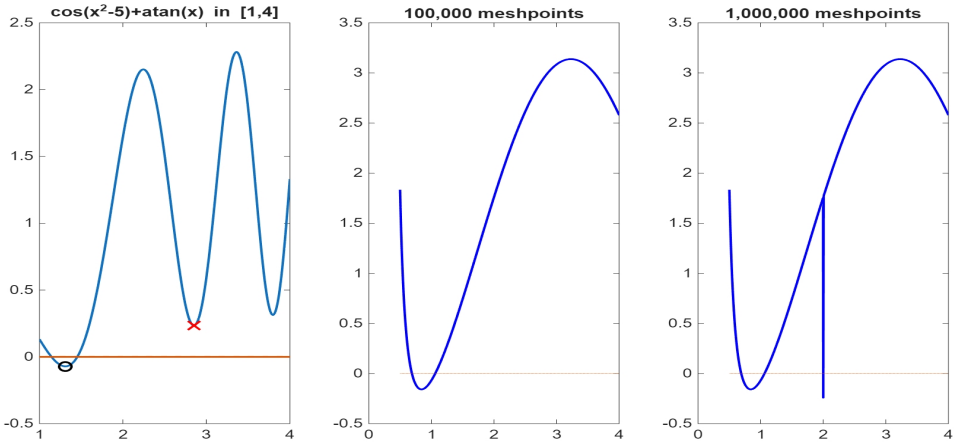
Taylor arithmetic is used, for example, in "`verifyquad`" for verified quadrature using a Romberg scheme with remainder term, cf. [22, Algorithm 12.2]:

```
>> I = verifyquad(f,3.5,4.5); fprintf('integral = %s\n',disp_(I))
integral =    0.38985530341309_
```

More examples and details are in [9, Demo 1, 2, 8, 12, 14, 15, 16 and 18].

# 9 Global (un-)constrained optimization

A key feature of INTLAB is the possibility to evaluate a function $f$ over some domain $X$ such that $\{f(x) : x \in X\} \subseteq f(X)$ is correct with mathematical certainty. In floating-point arithmetic we may evaluate a function using some mesh, but a minimum might hide between mesh points regardless how fine the mesh is: A result which is true under any circumstances is hardly possible. Consider $f(x) = \cos(x^2 - 5) + \operatorname{atan}(x)$ over $[1, 4]$. Matlab's function `fminbnd(f,1,4)` tries to find the minimum of $f$ over $X$ and identifies $\hat{x} = 2.85$ (red cross in the left graph) as the minimum, whereas




INTLAB proves that the true result is $\hat{x} = 1.31$ (black circle). That is verified by `verifyglobalmin(f,infsup(1,4))` with mathematical certainty. Plots of the function $g(x) := \Gamma(3\operatorname{atan}(x) - \cosh(\operatorname{erf}(x))) - 2\cos(x) - 2e^{-(10^6(x-2))^2}$ over the interval $X = [0.5, 4]$ are shown with 100,000 mesh points in the middle, and 1 million mesh points on the right, respectively. Using `verifyglobalmin(f,infsup(.5,4))` verifies the global minimum near $\hat{x} = 2$, in floating-point that spike is missed and Matlab's `fminbnd` claims that the minimizer is near 0.84.

We come back to Trefethen's problem in Section 3. For $X = [-10^6, 10^6]^2$ the call

```
>> [mu,L,Ls,data] = verifyglobalmin(@Tref,midrad(0,1e6)*ones(2,1));
   mu, L, sizeLs = size(Ls)
intval mu =
[   -3.4699,   -3.1447]
intval L =
   -0.3957
   -0.0235
sizeLs =
     2   150
```

produces an inclusion $\hat{x} = [-0.3957; -0.0235]$ of a local minimum with function value $\mu \in [-3.4699, -3.1447]$ over the search box $\pm 10^6$ in each dimension, and in addition

a list `Ls` of 150 potential candidates. The global minimum is verified to be in `L` or `Ls`. All necessary data to proceed further is stored in `data`, and the second call

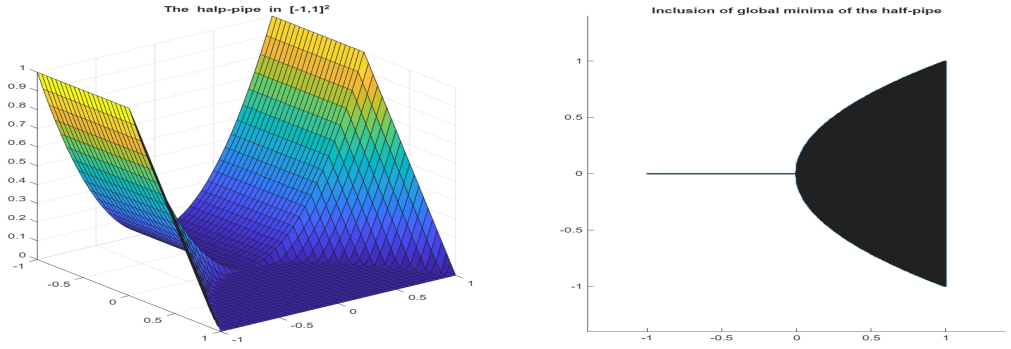
```
>> [mu,L,Ls,data] = verifyglobalmin(data); mu, sizeLs = size(Ls)
intval mu =
   -3.144763005605__
sizeLs =
     0     0
```

verifies the minimum value $\mu \in -3.144763005605 \pm 10^{-13}$, and because the list `Ls` of candidates is empty, that is mathematically verified to be the (only) global minimum.

The main problem in verified global optimization is a clever bisection strategy and how to discard subboxes which cannot contain a global minimum. To that end we use the methods in [25]. A scalable problem is the famous Griewank function $G : \mathbb{R}^n \to \mathbb{R}$ with $G(x) = 1 + \frac{1}{4000}\sum_{k=1}^{n} x_k^2 + \prod_{k=1}^{n} \cos\left(\frac{x_k}{\sqrt{k}}\right)$ on $X := [-600, 600]^n$. For $n = 50$ the function has some $10^{129}$ stationary points in $X$. The global minimizer is the origin independent of $n$. For $n = 50$, `verifyglobalmin` needs 12 seconds to produce a list `Ls` with 50 remaining candidates for the minimizer, and the second call verifies after 9 seconds that the (only) global minimum is $\hat{x} = 0$ with $G(\hat{x}) = 0$.

In Section 12 we treat overdetermined nonlinear systems, an example for a constrained optimization problem.

A famous optimization problem is the half-pipe $h(x, y) = \max(y^2 - \max(x, 0), 0)$ on $[-1, 1]^2$. The set of global minima is $M := \{(x, 0) : x \le 0\} \cup \{(x, y) : 0 \le x \le y^2\}$. The




function is not everywhere differentiable, so care is necessary when using algorithmic differentiation (cf. Section 8). The left picture shows the function on $X = [-1, 1]^2$, and the right one the set of minimizers. The call

```
>> [mu,L,Ls,data] = verifyglobalmin(h,infsup(-1,1)*ones(2,1)
```

needs 0.1 seconds to produce a first list of candidates, and two subsequent calls of `verifyglobalmin(data)` need another 2.5 seconds to produce a list `Ls` of 83,908 candidates. In that example the set $M$ of minima has non-empty interior, and to prove $x \in M$ is an ill-posed problem. Therefore, INTLAB computes a list of candidate inclusions covering $M$, the list `L` with verified minima must be empty.

More examples and details are in [9, Demo 1, 3, 4, 8, 12, 14 and 15].

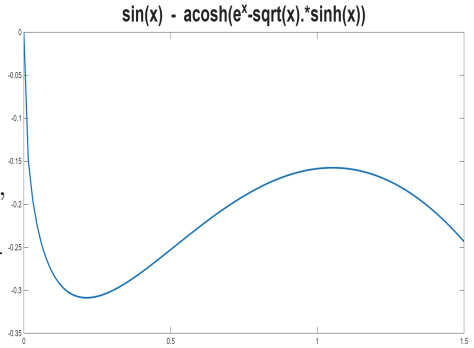
# 10 Ignore input out of range: A secret option

The function $f(x) := x^2 - \mathrm{acos}(\sin(x)^2 + \cos(x)^2)$ is equal to $x^2$ for all real $x$. Due to inevitable rounding errors, $\sin(\mathbf{X})^2 + \cos(\mathbf{X})^2$ contains values larger than 1 for any interval $\mathbf{X} \in \mathbb{IR}$ except $\mathbf{X} = [0, 0]$. As a consequence, for $f$ as a real function and any interval $\mathbf{X} \neq [0, 0]$ an inclusion of $f(\mathbf{X})$ cannot be evaluated.

To that end Arnold Neumaier proposed to ignore inputs out of range in certain situations. That may be useful in global optimization for wide input intervals. Consider

$$g(x) := \sin(x) - \mathrm{acosh}(e^x - \sqrt{x}\sinh(x))$$

The function is well defined on the interval $\mathbf{X} = [0, 1.5]$. The true range of the argument $e^{\mathbf{X}} - \sqrt{\mathbf{X}}\sinh(\mathbf{X})$ of the inverse hyperbolic cosine is is $[1, 1.8742]$, but interval evaluation results in $[-1.6079, 4.4817]$, and `g(X)` yields NaN. Using the option "*ignore input out of range*" the result of `g(X)` is $[-2.1805, 0.9975]$, an overestimation but correct inclusion of the true range $[-0.3088, 0]$, the latter computed by `verifyrange(g,X)`. That option is sometimes beneficial. It is used in INTLAB's optimization routines and the computation of all roots of a nonlinear system in a box, cf. Sections 9 and 12.




However, care is necessary. In Section 8 we introduced the interval Newton operator $N(\tilde{x}, \mathbf{X}) := \tilde{x} - f(\tilde{x})/f'(\mathbf{X})$ with the property that $N(\tilde{x}, \mathbf{X}) \subseteq \mathbf{X}$ proves that $\mathbf{X}$ contains a unique root of $f$ in $\mathbf{X}$ if $0 \notin f'(\mathbf{X})$ for an interval $\mathbf{X}$ and $\tilde{x} \in \mathbf{X}$.

Consider $h(x) := x^2/2 + 4x + 9 + 20\sqrt{x}$ with real range of definition $\mathbb{R}_{\geq 0}$. For $\mathbf{X} := [-4, 2]$ and $\tilde{x} := 1$ the interval Newton operator is not defined as $\mathbf{X}$ contains negative numbers. However, activating the option "*ignore input out of range*" we obtain $N(\tilde{x}, \mathbf{X}) = [-3.7377, 1.0000] \subseteq \mathbf{X}$ pretending that $h$ has a root in $\mathbf{X}$. That is not true as $h(0) = 9$ and $h$ is strictly monotonically increasing on its whole range of definition $\mathbb{R}_{\geq 0}$.

In Version 5.2 of INTLAB "*ignore input out of range*" was introduced as an option available to the user. However, examples like the previous one may happen unnoticed, resulting in false answers. Therefore, from Version 10 of INTLAB, the option is no longer available by INTLAB commands but only to expert users by explicitly accessing internal structures.

More precisely, the command `INTLAB_CONST.RealStdFctsExcptnIgnore = true` activates the "*ignore input out of range*" option, setting it false deactivates it again. To that end it is necessary to define `INTLAB_CONST` as a global constant. That constant with many subfields is ubiquitous in INTLAB, however, it should never be accessed directly by the user, only through the corresponding INTLAB routines.

In addition, the global variable `INTLAB_CONST.RealStdFctsExcptnOccurred` monitors whether some input of range was actually ignored since activating the option. In INTLAB's functions like optimization and all roots of a nonlinear system which use that option it is taken care by Matlab's "`onCleanup`" that after a sudden end, e.g. by Ctrl-C, the option is deactivated.

# 11 Affine arithmetic and range estimation

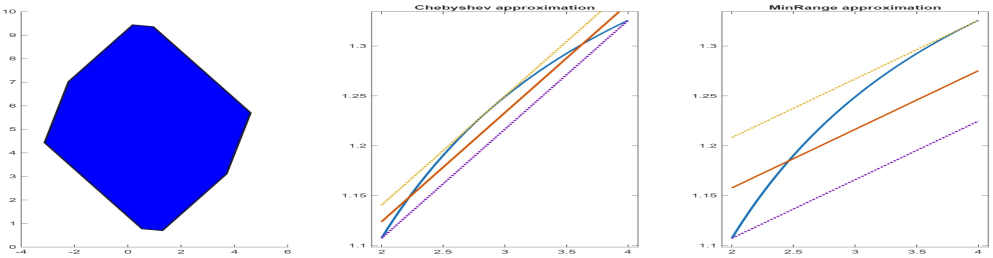
For $f : \mathbb{R}^m \to \mathbb{R}^n$ and an interval vector $X \in \mathbb{IR}^m$ the INTLAB call $Y = f(X)$ verifies that $f(x) \in Y$ for all $x \in X$. That is an important tool for many applications from optimization to solving systems of nonlinear equations. However, due to dependencies, there may be an overestimation, i.e., $Y$ may have larger, sometimes much larger diameter than $\{f(x) : x \in X\}$. A tight inclusion is in particular important to exclude regions $X$ in global optimization (cf. Section 9), i.e., to verify that $X$ cannot contain a global minimum and can be discarded.

There are several measures to fight overestimation, among them affine arithmetic based on [2] and with several improvements following [24]. Affine sets are represented by $c_0 + \sum c_i \mathbf{E}_i$ with $\mathbf{E}_i := [-1, 1]$. For example, `X = infsup(7,9)` and `Y = affari(X)` sets $c_0 = 8$ and $c_1 = 1$, and therefore `Y` $= \{8 + e : e \in [-1, 1]\}$. Consequently,

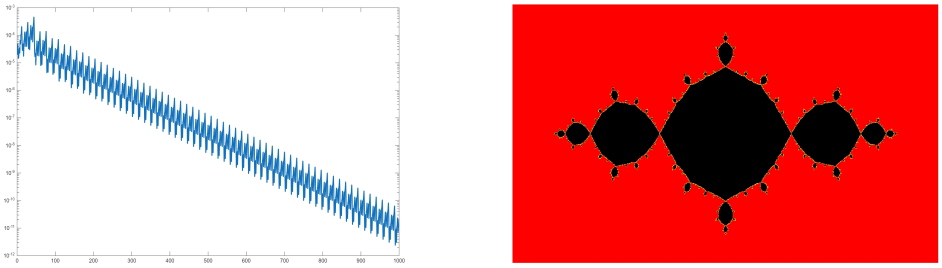
$$\texttt{Y} - \texttt{Y} = \{(8 + e) - (8 + e) : e \in [-1, 1]\} = 0 \ ,$$

whereas in interval arithmetic `X` $- $ `X` $= \{x_1 - x_2 : x_1, x_2 \in [-1, 1]\} = [-2, 2]$.

A 2-dimensional interval $X \in \mathbb{IR}^2$ is necessarily parallel to the axis, whereas `affari` sets are convex and bounded by parallel-epipeds. For example, the `affari` graph of




$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x - \sin(y) + x^y \\ \cos(y^x) - \sinh(x/y) \end{pmatrix}$ is shown on the left. The affari computation of standard functions has to modes, "Chebyshev" and "MinRange" approximation. The former reduces the area, the latter the range of the inclusion, see, for example, the graphs for atan in the middle and on the right.



Consider the famous Hénon map $\binom{x}{y} \mapsto \binom{1-ax^2+y}{bx}$, which is chaotic for $a = 1.06, b = 0.3$. We choose $a = 1.056$ and $b = 0.3$. Interval arithmetic for $X_0 := \binom{0}{0} \pm 10^{-5}$ produces $\pm\infty$ after 47 iterations. Using `affari` the maximal diameter of the entries of $X_k$ is plotted in the left graph. After 1,000 iterations the diameters are less then $10^{-10}$.

Define $z_{k+1} := z_k^2 + c$ for given $z_0 \in \mathbb{C}$ and fixed $c \in \mathbb{C}$. The Julia set is the boundary of the set $K_c$ for which the iteration remains bounded. We choose $c = -1$. It is known that the iteration diverges for $\|z_0\|_\infty \geq 2$. We iterate using a starting box $Z_0 \in \mathbb{IC}$ and the `affari` toolbox. The iteration is verified to be convergent if $Z_{k+\ell} \subseteq Z_k$ for integers $k, \ell$, and divergent if $\|z\|_\infty \geq 2$ for all $z \in Z_k$. The right graph shows values of $z_0$ with convergent iteration in black, with divergent in red, and if not decided after 100 iterations in yellow. The yellow area is tiny, almost invisible, otherwise that is a verified image of the filled Julia set.

Consider $f(x) = \sqrt{|x+5|} - (x-5)e^{x/10}$ over the interval $X = [-4, 1]$ with range $R = [6.87, 7.47]$. The following code computes `yint` using ordinary interval arithmetic,

```
>> f = inline('sqrt(abs(x+5))-(x-5).*exp(x/10)');
   xs = -4; X = infsup(-4,1);
   yint = f(X)
   yg = f(gradientinit(X)); ygrad = f(intval(xs)) + yg.dx*(X-xs)
   S = slopeinit(xs,X); ys = f(S); yslope = ys.r
   yaffine = f(affari(X))
   [ Yout , Yin ] = verifyrange(f,infsup(-4,1));
   fprintf('%6.4f <= min(f) <= %6.4f\n',Yout.inf,Yin.inf)
   fprintf('%6.4f <= max(f) <= %6.4f\n',Yin.sup,Yout.sup)
intval yint =
[    3.6812,   12.3961]
intval ygrad =
[    3.8682,   11.1546]
intval yslope =
[    6.4714,    9.0445]
affari yaffine =
[    6.6941,    8.2723]
6.8699 <= min(f) <= 6.8703
7.4692 <= max(f) <= 7.4698
```

i.e., replacing every operation by the corresponding interval operation. The result `yint` contains but overestimates the true range $R$.

For $\tilde{x} \in X$ it follows $f(X) := \{f(x) : x \in X\} \subseteq f(\tilde{x}) + f'(X)(X - \tilde{x})$. That is called the midpoint rule. The result `ygrad` for $\tilde{x} = -4$ is a better inclusion of $R$.

Using slopes (cf. Section 8) produces `yslope`, and `yaffine` the result of affine arithmetic.

An accurate inclusion of the range of $f(X) := \{f(x) : x \in X\}$ for $f : \mathbb{R}^m \to \mathbb{R}^n$ is obtained by INTLAB's "`verifyrange`". The algorithm is based on optimization methods using bisection. It allows to calculate an inner inclusion as in Section 5 and, if the result is not satisfactory, to improve the inclusion as in Section 9 by

```
>> [Yout,Yin,X,Data] = verifyrange(f,X);
   for k=1:kmax, [Yout,Yin,X,Data] = verifyrange(Data); end
```

Yet another, very effective methods are Taylor models, cf. Section 16.

In the example above affine arithmetic produced the tightest inclusion, however, there is neither a panacea nor one method being always superior. Fighting overestimation is a major challenge of verification methods.

For much more examples and details see [9, Demo 1, 13, 15, 16, 17 and 20].

## 12 Nonlinear systems and all roots

In [1] the discretization of $3y''y + (y')^2 = 0$ for $y_0 = 0, y_1 = 20$ is a test function for nonlinear systems with initial approximation $(10, \ldots, 10)^T$. Here we use $n = 3,000$.

```
>> sparsegradient(inf);    % gradient always full
   tic, Xfull = verifynlss(@test,xs); tfull = toc;
   tic, Xsparse = verifynlssSparse(@test,xs); tsparse = toc;
   fprintf('tfull = %5.2f, tsparse = %5.2f\n',tfull,tsparse)
   r = relerr(Xfull); relerrfull = [median(r) max(r)]
   r = relerr(Xsparse); rererrsparse = [median(r) max(r)]
tfull =  9.78, tsparse =  0.49
relerrfull =
   1.0625e-16   4.3735e-16
rererrsparse =
   1.8691e-12   9.7500e-10
```

The true solution is $y = 20x^{3/4}$. The command `sparsegradient(N)` specifies that gradients of functions with $N$ or more unknowns are stored as sparse matrices, similarly for `sparsehessian(N)`. Thus, the first command tells Algorithm `verifynlss` to treat Jacobians as full matrix, i.e., `verifylss` as in Section 5 is used. The function `verifynlssSparse` uses sparse Jacobians and a sparse linear system solver as in Section 6, the same as setting `sparsegradient(0)` and using Algorithm `verifynlss`.

Using full Jacobians takes 20 times as much computing time as `verifynlssSparse`, but is more accurate and robust. For $n = 10,000$ `verifynlss` needs 225 seconds to compute almost maximally accurate verified inclusions, but `verifynlssSparse` fails.

In Section 9 the global minimum of the Griewank function $G : \mathbb{R}^n \to \mathbb{R}$ with $G(x) = 1 + \frac{1}{4000}\sum_{k=1}^{n} x_k^2 + \prod_{k=1}^{n} \cos\left(\frac{x_k}{\sqrt{k}}\right)$ on $X := [-600, 600]^n$ was considered. It takes about 20 seconds to verify the global minimum for $n = 50$ unknowns, for $n = 2$ it takes 0.25 seconds. For $n = 50$ there is an estimated number of $10^{129}$ stationary points, and we may ask about the exact number in $X$ for $n = 2$.

```
>> X = infsup(-600,600)*ones(2,1);
   tic, [L,Ls,data] = verifynlssderivall(@Griewank,X); t1 = toc;
   tic, [L,Ls,data] = verifynlssderivall(data); t2 = toc;
time   206.7,  #elts(L)  206281 , #elts(Ls)  1
time     2.8,  #elts(L)  206281 , #elts(Ls)  0
```

The function `[L,Ls] = verifynlssderivall(f,X)` computes all roots of $f'$ within $X$, where the list `L` is a list of verified inclusions of roots of $f'$, and `Ls` is a list of candidates.

The first call verifies that the Griewank function has 206,281 stationary points within $X$ plus one not yet decided candidate. The second call builds upon the already computed data and verifies that the candidate in `Ls` does not contain a root of $f'$. As a consequence, there are totally 206,281 stationary points of $f$ in $X$.

By principle, verification methods cannot solve ill-posed problems. Therefore, it is not possible to verify that a matrix is singular or that a nonlinear system has a double root. For the latter we introduce some regularization [22]. Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be given. Then `verifynlss2` computes $\mathbf{X} \in \mathbb{IR}^n$, $\mathbf{E} \in \mathbb{IR}$ and $k$ such that there is $\hat{x} \in \mathbf{X}$

and $\hat{e} \in \mathbf{E}$ with $g(\hat{x}) = 0$, where $g : \mathbb{R}^n \to \mathbb{R}^n$ is the same as $f$ except that the $k$-th component function $f_k : \mathbb{R}^n \to \mathbb{R}$ is replaced by $g_k(x) = f_k(x) - \hat{e}$. As an example, we modify Brown's almost linear function $f : \mathbb{R}^n \to \mathbb{R}^n$ into $f_k(x) = x_k + \sum_{i=1}^{n} x_i - (n+1)$ for $1 \le k \le n-1$ and $f_n(x) = \prod_{i=1}^{n} x_i - \frac{n+1}{n}(1 - \frac{1}{n^2})^{n-1}$. Then the Jacobian at $(\alpha, \ldots, \alpha, 1 + \frac{1}{n})^T$ with $\alpha = 1 - \frac{1}{n^2}$ is singular. An inclusion for $n = 1000$ is obtained after 110 seconds. The initial approximation is $(1, \ldots, 1)^T$, and we show the first and last 3 entries of $\mathbf{X}$, as well as $\mathbf{E}$. Here $k = n$, i.e., the last equation was regularized.

```
>> n = 1000, p = (1-1/n^2)^(n-1)*(1+1/n)-1;
   [X,xs,k] = verifynlss2(@Brown,ones(n,1),-1,0,p);
   format long, XX = X([1:3 n-2:n]); disp_(XX)
   format short, E = X(n+1); infsup(E)

intval XX =                                    intval E =
   0.9999990000____     0.9999990000____          1.0e-010 *
   0.9999990000____     0.9999990000____          [  -0.0453,   0.1455]
   0.9999990000____     1.001000000000__
```

Next we come to overdetermined nonlinear systems. We use a GNSS (Global Navigation Satellite System) positioning with clock bias $b$ as a model problem. We use 8 measurements (pseudoranges) modelled as

$$r_i = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2} + b + \varepsilon_i, \quad i = 1, \ldots, 8$$

with known satellite positions $(x_i, y_i, z_i)$ and additive Gaussian noise $\varepsilon_i \sim \mathcal{N}(0, 0.01^2)$. In our scenario, the satellites are predominantly planar but not exactly coplanar, resulting in a moderately ill-conditioned Jacobian. The true receiver state is $(1.2, -0.8, 0.5, 0.07)^T$. The following code computes an approximate result by Matlab's "`lsqnonlin`", and based on that a verified inclusion by INTLAB:

```
>> xs = lsqnonlin(f,zeros(4,1)), X = verifynlssnonsquare(f,xs)

xs =                        intval X =
   1.199168259413428           1.19916928148273
  -0.801958621334932          -0.8019384050255_
   0.688093264796084           0.68731372012___
   0.046391311355202           0.046500948748__
```

The inclusion is accurate, the approximation has some 3 to 6 correct digits.

The same problem with only 3 measurements results in an underdetermined nonlinear system $f : \mathbb{R}^3 \to \mathbb{R}^4$. Amongst $f(x) = 0$ we seek for minimal $\|x\|_2$.

```
>> xs = lsqnonlin(f,zeros(4,1)), X = verifynlssnonsquare(f,xs)
   residual_xs = norm(f(xs)), residual_Xmid = norm(f(X.mid))

xs =        residual_xs =      intval X =           residual_Xmid =
   1.0625      4.9896e-14        1.05347798383568      8.8818e-16
  -0.6575                      -0.64074475879471
   0.1865                       0.26639944555375
  -0.0733                      -0.04653710701576
```

The approximation $\tilde{x}$ is good, however, with verification we find $\mathbf{X}$ including a point with smaller residual. The example illustrates the intrinsic non-uniqueness of GNSS positioning with an unknown clock bias when fewer than four satellites are available.

More examples and details are in [9, Demo 4, 8, 12, 14 and 18].

## 13 Eigenproblems and clustered or multiple roots

The algebraic eigenproblem $Ax = \lambda x$ can be rewritten as $f(x, \lambda) = Ax - \lambda x$ plus an additional normalization equation for the eigenvector. That results in a nonlinear system with $n + 1$ unknowns and can be solved with the methods in the previous section. We generate a $500 \times 500$ matrix and calculate an inclusion of the first eigenpair:

```
>>  n = 500; A = randmat(n,1e12);
    tic, [v,d] = eig(A); k = 1;
         XL = verifynlss(@nlsseig,[v(:,k);d(k,k)],[],[],A); toc
    rX = relerr(XL(1:n)); [median(rX) max(rX) relerr(XL(n+1))]
Elapsed time is 10.995088 seconds.
ans =
    2.2231e-07   2.8733e-06   3.4835e-09
```

The result is not bad, however, the general nonlinear system solver requires quite some computing time. Using `[L,X] = verifyeig(A,d(1),v(:,1))` in the code above calls INTLAB's routine `verifyeig` to compute an inclusion of the eigenpair. The result is

```
Elapsed time is 0.114063 seconds.
ans =
    2.5288e-13   4.0602e-10   3.7535e-14
```

This is faster by two orders of magnitude and more accurate. However, `verifyeig` computes verified bounds for each eigenpair (or cluster) separately. From Version 14 of INTLAB we use the algorithm in [26] to compute inclusions of all eigenpairs (in order to save space, the computation of the relative errors is omitted):

```
>> n = 500; A = randmat(n,1e12);
   tic, [v,d] = eig(A);
   for k=1:n, [L,X] = verifyeig(A,d(k,k),v(:,k)); end
   tverifyeig = toc
   tic, [XX,LL] = eig(intval(A)); teig = toc
tverifyeig =
   71.9300
relerrLX =
   7.8677e-08   2.0356e-03   5.0568e-07   1.3743e-02
teig =
   2.5348
relerrLLXX =
   3.3843e-16   1.0994e-13   3.8844e-16   3.7951e-08
```

Output are the median and maximum relative errors of the eigenvalues (first two numbers) and eigenvectors (last two). The verified version of Matlab's `eig` is not only faster than computing individual bounds for each eigenpair but also more accurate due to the extensive use of the new matrix residual algorithm `prodK` as in Section 4.

Next we consider eigenvalue clusters and/or multiple eigenvalues. To that end we use the well known Wilkinson test matrix, being symmetric tridiagonal. Most famous

```
>> n = 21; W = wilkinson(n);  % tridiagonal symmetric
   v = 1:4; W_NW = W(v,v)
   v = n-3:n; W_SE = W(v,v)
```

```
W_NW =                          W_SE =
10     1     0     0            7     1     0     0
 1     9     1     0            1     8     1     0
 0     1     8     1            0     1     9     1
 0     0     1     7            0     0     1    10
```

is the case $n = 21$ with one negative and 10 eigenvalue clusters of size 2. We first use `[L,X] = verifyeig(W,d(k,k),v(:,k))` to compute inclusions of the $k$-th eigenpair of the Wilkinson matrix `W` and display the inclusions of the first (left) and last (right) 4 eigenvalues:

```
-1.12544152211998        9.21067864730491
 0.25380581709667        9.21067864736133
 0.94753436752929       10.74619418290332
 1.78932135269508       10.74619418290339
```

All eigenvalue inclusions are maximally accurate. Eigenvalues of a symmetric matrix are perfectly well conditioned, but eigenvectors may be not. Following is the inclusion of the most sensitive eigenvector to $\lambda_{21}$ together with its diameter, on the left of the first and on the right for the last 4 entries, respectively:

```
   0.56___________    1.622e-03       0.04805373844125_    3.539e-16
   0.42___________    1.214e-03       0.16949775589369_    1.388e-16
   0.169__________    4.904e-04       0.41742001280919_    1.110e-16
   0.048__________    1.396e-04       0.55939864230118_    0.000e+00
```

The sensitivity of the eigenvector is of the order $2^{-53}\|\mathtt{W}\|/|\lambda_{20} - \lambda_{21}| \approx 0.017$, and that is reflected by the eigenvector inclusions. The algorithm `verifyeig` allows to treat eigenvalues as clusters by specifying the approximate subspace `v(:,20:21)`:

```
>> [Lpair,Xpair] = verifyeig(W,d(21,21),v(:,20:21));  Lpair

intval Lpair =
10.7461941829034_ +  0.0000000000000_i
```

It is verified that `Lpair` contains exactly 2 eigenvalues with an inclusion `Xpair` of the corresponding invariant subspace. To save space the inclusion of `Xpair` is not displayed as it is maximally accurate. As a mathematical principle, the inclusion of the eigenvalue pair must be complex: If the 2 true eigenvalues would be real and equal, a tiny perturbation of $A$ may move them into the complex plane. The verified version of Matlab's `eig` in INTLAB allows to specify the treatment of clusters as well:

```
>> [V,D] = eig(intval(W));  [Vcl,Dcl] = eig(intval(W),'kappa',1e-14);
[  0.5194,  0.5994] [  0.4916,  0.5869]     0.5594_    0.5392_
[  0.3774,  0.4574] [  0.3547,  0.4501]     0.4174_    0.4024_
[  0.1295,  0.2095] [  0.1157,  0.2111]     0.1695_    0.1634_
...
[ -0.2034, -0.1234] [  0.1218,  0.2172]    -0.1634_    0.1695_
[ -0.4424, -0.3624] [  0.3698,  0.4651]    -0.4024_    0.4174_
[ -0.5793, -0.4993] [  0.5117,  0.6071]    -0.5392_    0.5594_
```

All eigenvalue inclusions are maximally accurate and not displayed. With increasing magnitude of the eigenvalue, the eigenvectors become more sensitive and the inclusions wider. We display above in the first and second column the first and last 3 entries of the eigenvectors to $\lambda_{20\ldots21}$ for the first call `eig(intval(W))`. They are very wide

because individual eigenpairs are included. The second call specifies that eigenvalues closer than $\kappa = 10^{-14}$ are treated as a cluster. That regularizes the problem as shown in third and fourth column. The display due to space limitation camouflages that all inclusions are almost maximally accurate. The inclusions to the first call differ because those on the left are for eigenvectors, those on the right for the basis of an invariant subspace. The next example is on a multiple defective eigenvalue. Consider

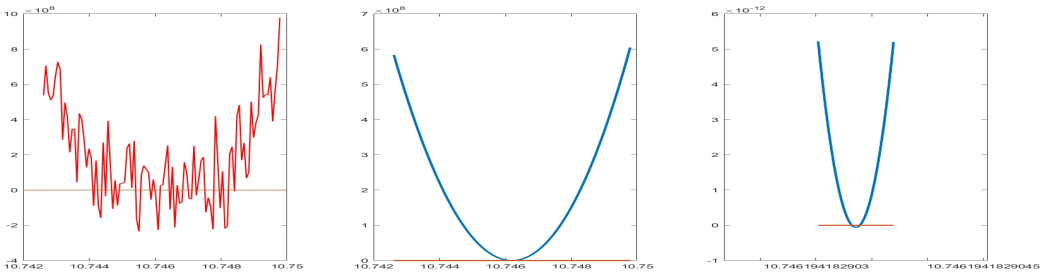
```
>> n = 4; A = diag(1:n);
   k = 3; A(1:k,1:k) = band(ones(k),0,1);
   M = randmatdet1(n); B = M*A*round(inv(M));
```

The matrix $A$ is upper triangular and has a $3 \times 3$ Jordan block to the eigenvalue 1 in the upper left, plus an eigenvalue 4 in the lower right corner. The matrix $M$ is an integer matrix with determinant 1, and it follows $M^{-1} =$ `round(inv(M))` taking care of rounding errors for the integer matrix $M^{-1}$. As a consequence, the Jordan structure of the matrices $A$ and $B$ are identical, i.e., both have a 3-fold defective eigenvalue 1 and an additional eigenvalue 4. The following code specifies the approximation 1 for the eigenvalue and a 3-dimensional approximation of the invariant subspace:

```
>> [V,D] = eig(B); [L,X] = verifyeig(B,1,V(:,2:n))
intval L =
   1.000_____________ -  0.000_____________i
intval X =
  -0.3779 -  0.0001i   -0.3779 +  0.0001i   -0.3779 +  0.0000i
   0.3779 -  0.0000i    0.3779 +  0.0000i    0.3779 +  0.0000i
  -0.3779 -  0.0000i   -0.3779 +  0.0000i   -0.3779 +  0.0000i
   0.7559 +  0.0000i    0.7559 +  0.0000i    0.7559 +  0.0000i
```

The sensitivity of the eigenvalue is $(2^{-53})^{1/3}$, reflected in the accuracy of the inclusion.

One may calculate the roots of a polynomial by the eigenvalues of the companion matrix, but not eigenvalues of a matrix by the roots of the characteristic polynomial. The command `poly(W)` computes the characteristic polynomial of the Wilkinson



matrix. Its coefficients are integers, and one can verify that `P = round(poly(W))` is indeed equal to $\chi_W$. An inclusion of the roots near $\lambda_{20\ldots21}$ is calculated by

```
>> r = roots(P); [X,k,e] = verifypoly(polynom(P),r(1))
```

The computed inclusion is `X = 10.75__`, where the output $k = 2$ and $e = 1$ verifies that there are exactly 2 roots of $P$ in `X`. The graph using floating-point on the left, and INTLAB's `long` arithmetic with 50 decimals in the middle show the ill-condition of the two roots. The right graph zooms into `X.mid`$\pm 3 \cdot 10^{-13}$ and decovers that there are 2 real roots of P. More examples and details are in [9, Demo 3, 7, 12, 13, 14 and 15].

# 14 Parameter identification

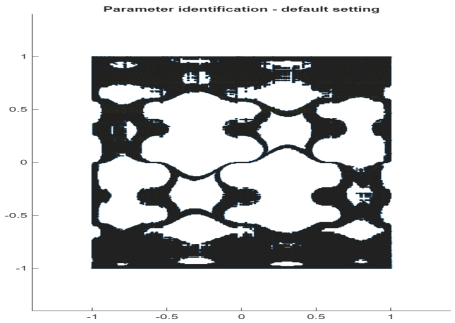
A function $f : \mathbb{R}^m \to \mathbb{R}^n$ with $m > n$ may have an $(m-n)$-dimensional zero set which can be identified by parameter identification. Consider

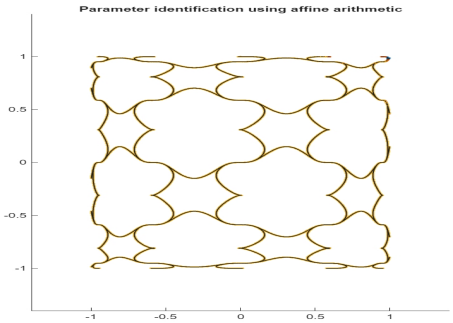
$$f(x,y) = (5y - (5x - 20x^3 + 16x^5)^3 - 20y^3 + 16y^5)^2 - (5y - 20y^3 + 16y^5)^6$$

on $X = [-1,1]^2$. The following code visualizes the zero set, left using interval arithmetic and right using affine arithmetic.

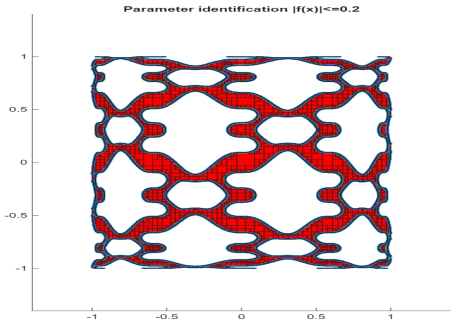
```
>> param = verifynlssparamset('Display','~');
   X = infsup(-1,1)*ones(2,1);
   tic, verifynlssparam(f,0,X,param); tIntval = toc;
   param.Method = 'affari';
   tic, verifynlssparam(f,0,X,param); tAffari = toc;
   fprintf('tIntval = %3.1f, tAffari = %3.1f\n',tIntval,tAffari)
tIntval = 0.3, tAffari = 5.3
```

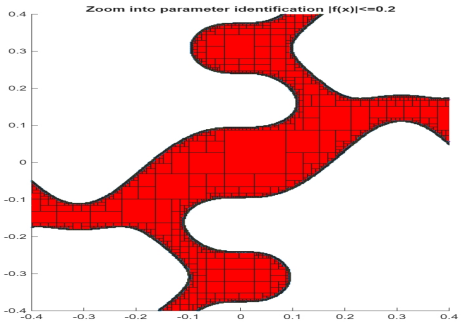
The function $f$ is verified to be nonzero in the white area, and potentially zero in the black area. It can be verified that the function values are nonzero over some box in $X$, however, checking $f(x_1,x_2) = 0$ for $(x_1,x_2) \in X$ is an ill-posed problem and cannot be verified (as in Section 9 for the half-pipe).




The affine arithmetic gives a pretty clear image of the zero set. Using `verifynlssparam(f,midrad(0,0.2),X)` computes the set of points $Y$ in $X$ with $|f(y_1,y_2)| \le 0.2$. Now $Y$ has a non-empty interior, and the set of points which are




verified to be in $Y$ are printed in red. The left graph is over $X = [-1,1]^2$, the right graph zooms into $[-0.4,0.4]^2$ and shows the size of boxes verified to be contained in $Y$.

More examples and details are in [9, Demo 1, 7, 12 and 19].

# 15 Matrix decompositions and structured problems

From Version 14, INTLAB offers a number of verified matrix decompositions based on [27], among them `lu, svd, eig, qr, chol, schur, polar` and `takagi`. Based on those, verified bounds for a number of other standard problems in numerical analysis can be computed. For example, the Procrustes problem asks for orthogonal $Q$ with minimal $\|AQ - B\|_F$ for given $A, B$ and has the solution $Q = UV^T$ for $[U, S, V] = \mathrm{svd}(A^T B)$. For $A = I$ this is the nearest orthogonal matrix to $B$. The smallest 2-norm distance of given $A$ to a singular matrix $B$ is $\sigma_{\min}(A)$, and an inclusion of a minimizer `B` is computed by `[U,S,V]=svd(intval(A)); S(end)=0; B=U*S*V'`.

The basis for computing an interval matrix $\mathbf{E}$ such $\det(A + E) = 0$ for $E \in \mathbf{E}$ by Algorithm "`verifyrankpert`" is given in [12].

In Section 5 we defined the outer and inner inclusions of the solution set $\Sigma(\mathbf{A}, b) := \{x \in \mathbb{R}^n : \exists A \in \mathbf{A}, Ax = b\}$ for a linear system with an interval matrix $\mathbf{A}$. However, a potential structure of the matrix $\mathbf{A}$ is ignored. Consider the discretization of the 1-dimensional Laplace operator with matrix $mA := \begin{pmatrix} -2 & 1 & 0 & \dots \\ 1 & -2 & 1 & \dots \\ 0 & 1 & -2 & \dots \\ & \dots & & \end{pmatrix}$ where each nonzero entry is afflicted with a tolerance $\pm 2 \cdot 10^{-7}$ and right hand side such that the solution of the midpoint system is $(1, -1, \dots)^T$. The following code[2] computes for $n = 1,000$ outer and inner inclusions of $\Sigma(\mathbf{A}^{\mathrm{struct}}, b)$ where $\mathbf{A}^{\mathrm{struct}}$ is the set of matrices $A \in \mathbf{A}$ according to the given structure, namely symmetric, Toeplitz and symmetric Toeplitz.

```
>> n = 1000; e = midrad(ones(n,1),2e-7);
   A = full(spdiags([e -2*e e],-1:1,n,n));
   xs = ones(n,1); xs(2:2:n) = -1; b = A.mid*xs;
   tic, [x,xinf,xsup] = verifylss(A,b); t = toc;
   fprintf(' ... ','general',t,max([diam(x) abs(xsup-xinf)]))
   for k=1:3
     switch k
       case 1, struct = 'symmetric';
       case 2, struct = 'generalToeplitz';
       case 3, struct = 'symmetricToeplitz';
     end
     tic, [y,yinf,ysup] = verifystructlss(structure(A,struct),b);
     fprintf(' ... ',struct,toc,max([diam(y) abs(ysup-yinf)]))
   end
         general  time   0.2   outer 0.21861724  inner  0.18218196
       symmetric  time  77.7   outer 0.10962151  inner  0.09117809
  generalToeplitz time   6.8   outer 0.00000080  inner  0.00000080
symmetricToeplitz time   5.1   outer 0.00000045  inner  0.00000040
```

For general and all structured matrices inclusions bear not much overestimation as the difference between the outer and inner inclusion is relatively small, respectively. The structure "symmetry" implies relations between $n(n-1)/2$ elements, the reason for the larger computing time. With Toeplitz structure the diameter of the solution set shrinks substantially. More examples and details are in [9, Demo 1, 3, 7 and 19].

[2] The output format is omitted to fit the `fprintf` statements into one line.

# 16 Verified solution of ordinary differential equations

The are two packages within INTLAB for the verified solution of ordinary differential equations, both designed and written by Florian Bünger. The AWA toolbox is based on [14], and the Taylor model toolbox on [4, 15, 16] and the literature cited over there. For much information and details see [9, Demo 10 and 11]; due to space limitations we can give here only few examples.

For the AWA toolbox consider the van der Pool equation $y'' - c(1-y^2)y' + y = 0$. Executable code for the rewritten system as first-order ODEs is

```
function dydt = vdp_fun(t,y)
  dydt = [ y(2) ; (1-sqr(y(1))).*y(2)-y(1) ];
```

with Jacobian

```
function dydt = vdp_fun(t,y)
  dydt = [ y(2) ; (1-sqr(y(1))).*y(2)-y(1) ];
```

The AWA toolbox offers the possibility to specify interval initial conditions, here $y_0 = (3 \pm 0.001) \times (-3 \pm 0.001)$. Then the call

```
>> [T,Y] = awa(@vdp_fun,@vdp_jac,[0,10],midrad([3;-3],1e-3));
```

produces a floating-point column vector of time points `T` in the integration interval $[0, 10]$ and a solution interval array `Y`, where each row in `Y` corresponds to a time returned in the corresponding row of `T`. The first column of `Y` contains inclusions for $y_1$, and the second column for $y_2$. A plot of the solution is as follows. The left graph

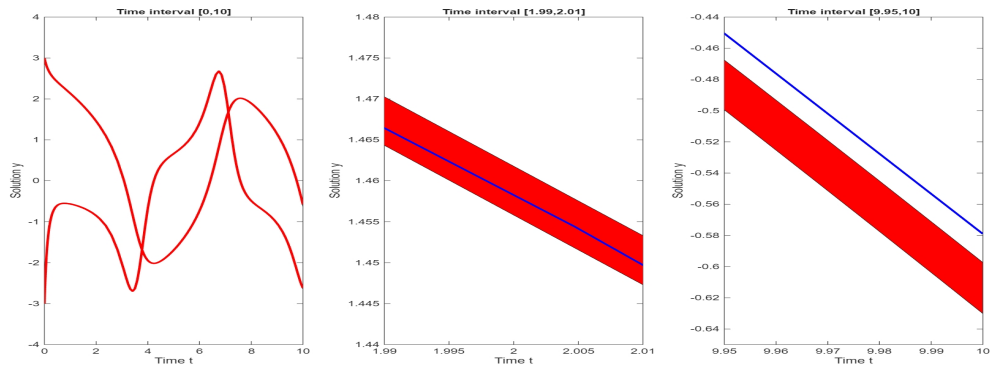
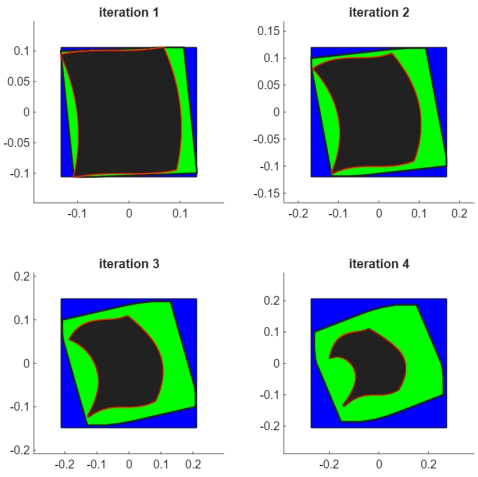


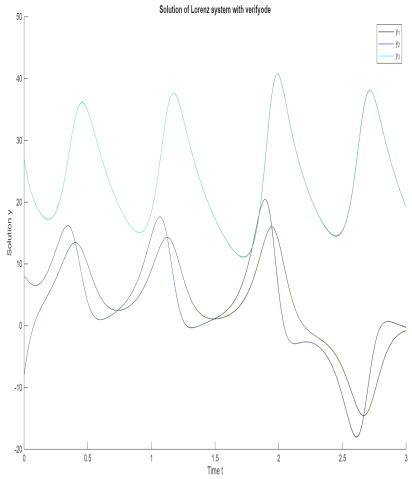

**Figure 1** Solution of the van der Pol Equation with INTLAB-AWA

is the continuous inclusion of the solution over the whole interval $[0, 10]$. Note that these are two curves for the lower and the upper bound of the solution which are visually identical. The middle and right graph zooms into the intervals $[1.99, 2.01]$ and $[9.95, 10]$. Now the lower and upper bound of the inclusions can be distinguished. The blue curve is the solution of Matlab's `ode45` with standard parameters, which is outside the inclusion interval in the right graph. With options for higher accuracy the approximation of `ode45` is within the validated bounds of `awa`.

The second toolbox for ODEs is based on Taylor models, see [9, Demo 11]. Taylor models are multivariate polynomials together with arithmetic operations and standard functions. A nice example of the power of Taylor model arithmetic by Florian

Bünger is the iteration $f : \mathbb{R}^2 \to \mathbb{R}^2$ with $f(x,y) = \begin{pmatrix} x - (0.125 + 2y)y \\ 6x^3 + y \end{pmatrix}$ over $B_0 := [-0.1, 0.1]^2$. The following graph shows $B_{k+1} := f(B_k)$ for $k = 1 \ldots 4$ on the left. The blue plot is interval arithmetic, the green affine and the black Taylor model arithmetic. The Taylor model result closely tracks the true domains $B_k$.

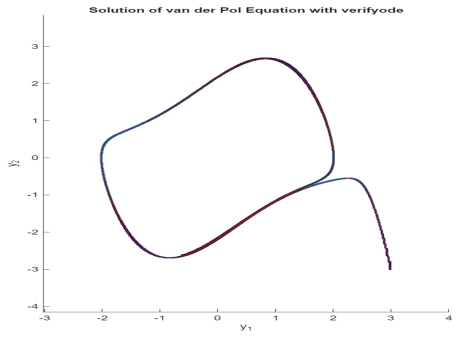


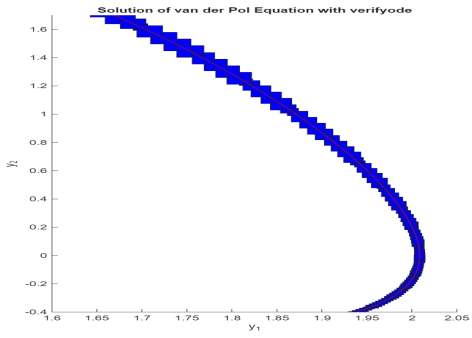




Another example is the Lorenz system

$$y_1' = \sigma(y_2 - y_1), \quad y_2' = (\varrho - y_3)(y_1 - y_2) \quad y_3' = y_1 y_2 - \beta a_3$$

with $\sigma = 10$, $\varrho = 28$ and $\beta = 8/3$. The verified inclusions are shown above on the right. There are a number of techniques to tighten the bounds such as QR preconditioning, blunting or shrink wrapping. For details cf. [4, 15, 16] and [9, Demo 11]. The inclusions guarantee some 2 correct figures for $y_{1\ldots3}$ over the whole domain.




Finally again the van der Pol equation above, now drawn as a verified phase portrait plotting $y_2$ against $y_1$. It shows in the zoom the overlapping boxes with the approximation by `ode45` in red.

Much more details can be found in [9, Demo 10 and 11] and in particular in [4] concerning shrink wrapping.

# 17 Other types of arithmetic

The Babylonian clay tablet YBC 7289 contains the remarkably accurate approximation $\sqrt{2} \approx (1; 24, 51, 10)_{60}$. In the toolbox "`flbeta`" a base up to $\beta = 64$ with precision $p$ can be specified, and an IEEE 754 conformant arithmetic is performed. Indeed, for base $\beta = 60$ the Babylonian approximation has 4 correct base-$\beta$ digits (8 decimals):

```
>> flbetainit(5,60,100); format flbetadigits;
   res_60 = sqrt(flbeta(2)), double_res = double(res_60)
flbeta-type res_60 =
+  1. 24 51 10  8 * 60^+000
double_res =
   1.414213580246914
```

Let a floating-point arithmetic with base $\beta$ and precision $p$ with relative rounding error unit $\mathbf{u}$ be given. For floating-point numbers $x, y$ denote by $\mathrm{fl}(x \circ y)$ the floating-point result of $t := x \circ y \in \mathbb{R}$. It is well known that the relative error $E_1(t) := \frac{|t - \mathrm{fl}(t)|}{|t|}$ is bounded by $\frac{\mathbf{u}}{1+\mathbf{u}}$. In [10, Theorem 2.1] it is shown that the bound is sharp if, and only if, $t := 1 + \mathbf{u}$ is attained as the result of $x \circ y$. In binary that is true if, and only if, $2^p + 1$ is not prime [10, Theorem 3.2], i.e., not a Fermat number.

```
>> for p=15:17
     flinit(p,100); succ1 = 1+2^(-p); x = flsequence(1,2);
     setround(1), y = double(1./x); setround(0)
     optimal = string(logical(any(double(x).*y==succ1)));
     fprintf('precision = %2d,   optimal bound  %s\n',p,optimal)
   end
precision = 15,   optimal bound  true
precision = 16,   optimal bound  false
precision = 17,   optimal bound  true
```

We check that using the `fl` toolbox emulating an IEEE 754 conformant binary arithmetic with specifiable precision $p$. The command `x = flsequence(a,b)` generates all floating-point numbers in precision $p$ in the interval $[a, b]$. The only candidate for $xy = 1 + \mathbf{u}$ is $y = 1/x$ in rounding upwards. The code checks $xy = 1 + \mathbf{u}$ for precision $15 \le p \le 17$. Indeed, the bound is not optimal for $p = 16$ as $2^{16} + 1$ is a Fermat prime.

There is a Galois field GF[p] toolbox `gfp` in INTLAB. One application is to verify that a given floating-point matrix is not singular, independent of the condition number.

```
>> n = 1000; A = randmat(n,1e30);
   tic, reg = isregular(A); t = toc;
   fprintf('regular = %2d,   time %5.3f seconds\n',reg,t)
regular =  1,   time 0.085 seconds
```

Regularity is verified by Gaussian elimination in GF[p] for $p = 3,001,171$. If $U_{nn} \neq 0$, then $A$ is verified to be regular. If $U_{nn} = 0$, then `isregular` may try another prime, decreasing the probability that still $U_{nn} = 0$ for regular $A$. Singularity of $A$ cannot be verified because this is an ill-posed problem, i.e., every epsilon-neighbourhood $U_\varepsilon(A)$ of $A$ contains a regular matrix.

More examples and details are in [9, Demo 3, 17, 19, 20, 21, 22, 23 and 24].

# 18 Finding files in INTLAB

There are more than 2000 m-files in INTLAB with more than 77,000 lines of code (without comments), and sometimes it is difficult to find the appropriate routine. To that end "`helpintlab str`" may be used. It searches all function definitions and second lines (short definition of a function) for the string `str`. For example,

```
>> helpintlab nonlinear

   1  TEST                 some nonlinear test functions
   2  VERIFYGLOBALMIN      Global nonlinear system solver
   3  VERIFYNLSS           Verified solution of nonlinear system
   4  VERIFYNLSS2          Verified solution of nonlinear system for double and multiple roots
   5  VERIFYNLSSSPARSE     Verified solution of nonlinear system
   6  VERIFYNLSSALL        Global nonlinear system solver
   7  VERIFYNLSSDERIVALL   Global nonlinear system solver for derivative
   8  VERIFYNLSSNONSQUARE  Verified solution of over- or underdeterminted nonlinear system
   9  VERIFYNLSSPARAM      Nonlinear system parameter estimation
```

gives an overview of functions involving some nonlinear problem. Since helpintlab searches for text strings, the call `helpintlab linear` would display nonlinear functions as well. To that end use

```
>> helpintlab ' linear'

   1  STRUCTLSS         Dense linear system solver for structured matrices
   2  VERIFYLSS         Verified solution of linear system
   3  VERIFYLSSSPARSE1  Solution of sparse linear systems, default version
   4  VERIFYLSSSPARSE2  Solution of sparse linear systems, second version
   5  VERIFYSTRUCTLSS   Dense linear system solver for structured matrices
   6  PLOTLINSOL        Plots solution set of real interval linear system in 2 or 3 unknowns
   7  BROWN             Brown's almost linear function
   8  SOLVEWPP          Solution of square linear system by GE with partial pivoting, generic routine
   9  SOLVEWTP          Solution of square linear system by GE with total pivoting, generic routine
```

If an additional parameter "0" is added, the path information is displayed as well:

```
>> helpintlab ill-co 0

   1  GENDOT      Generation of extremely ill-conditioned dot products
   2  GENSUM      Generation of extremely ill-conditioned sums
   3  RANDMAT     (ill-conditioned) random matrix, coefficients in [-1,1]
   4  INVILLCO    Inverse of extremely ill-conditioned matrices
   5  BOOTHROYD   Ill-conditioned matrix with checkerboard sign pattern
   6  ISFULLRANK  Proves that an ill-conditioned matrix A has full rank
   7  ISREGULAR   Proves that an ill-conditioned matrix A is regular

   1  accsumdot\GenDot.m
   2  accsumdot\GenSum.m
   3  random\randmat.m
   4  utility\InvIllco.m
   5  utility\boothroyd.m
   6  utility\isfullrank.m
   7  utility\isregular.m
```

The call "helpintlab str 1" displays the list of results found followed by a prompt to edit individual and/or all files:

```
>> helpintlab eig 1

   1  GERSHGORIN       Complex interval vector containing eigenvalues of matrix A
   2  STRUCTEIG        Verification of eigencluster near (lambda,xs) for structured input matrix
   3  VERIFYEIG        Verification of eigenvalue (cluster) near (lambda,xs)
   4  VERIFYEIGALL     Verified bounds for all eigenvalues and eigenvectors
   5  VERIFYSTRUCTEIG  Verification of eigencluster near (lambda,xs) for structured input matrix
   6  VERIFYSYMEIGALL  Verified eigendecomposition for symmetric/Hermitian matrix
   7  EIG              Verified bounds for all eigenvalues and eigenvectors
   8  GERSHGORIN       Complex interval vector containing eigenvalues of matrix A

   Press Enter to finish, or enter 0 to see file paths, or enter index vector of files to be opened.
```

# References


[1] J.P. Abbott and R.P. Brent. Fast local convergence with single and multistep methods for nonlinear equations. *Austr. Math. Soc. 19*, pages 173–199, 1975.

[2] M.V.A. Andrade, J.L.D. Comba, and J. Stolfi. Affine arithmetic. Presented at INTERVAL'94, 1994.

[3] D.H. Bailey. A Fortran-90 based multiprecision system. *ACM Trans. Math. Software*, 21(4):379–387, 1995.

[4] F. Bünger. Shrink wrapping for Taylor models revisited. *Numerical Algorithms*, 78(4):1001–1017, 2018.

[5] T.A. Davis, Y. Hu: The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software 38, 1, Article 1, 2011.*

[6] P. Holoborodko. *Multiprecision Computing Toolbox for MATLAB 5.4.3.16137.* Advanpix LLC., Yokohama, Japan, 2025.

[7] I. Gargantini and P. Henrici. Circular arithmetic and the determination of polynomial zeros. *Numer. Math.*, 18:305–320, 1972.

[8] IEEE Standard for Floating-point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[9] 25 INTLAB demos. `www.tuhh.de/ti3/rump/intlab/demos/html/index.html`

[10] C.-P. Jeannerod and S.M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Math. Comp.*, 87:803–819, 2017.

[11] M. Lange and S.M. Rump. Faithfully rounded floating-point operations. *ACM Trans. Math. Softw.*, 46, 2020.

[12] M. Lange and S.M. Rump. Verified inclusions for a nearest matrix of specified rank deficiency via a generalization of Wedin's $\sin(\theta)$ theorem [Winner of the Moore prize 2021]. *BIT*, 61:361–380, 2021.

[13] M. Lange and S.M. Rump. Accurate floating-point matrix residuals. to appear.

[14] R. Lohner. *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen.* PhD thesis, University of Karlsruhe, 1988.

[15] K. Makino. *Rigorous analysis of nonlinear motion in particle accelerators.* Ph.D. dissertation, Michigan State university, 1998.

[16] K. Makino and M. Berz. Suppression of the wrapping effect by Taylor model-based verified integrators: long-term stabilization by preconditioning. *International*

*Journal of Differential Equations and Applications*, 10(4):353–384, 2005.

[17] MATLAB. User's Guide, Pre-Release 2026a, the MathWorks Inc., 2025.

[18] S. Oishi, K. Ichihara, M. Kashiwagi, T. Kimura, X. Liu, H. Masai, Y. Morikura, T. Ogita, K. Ozaki, S. M. Rump, K. Sekine, A. Takayasu, N. Yamanaka: *Principle of Verified Numerical Computations*. Corona Publisher, Tokyo, 2018. [in Japanese].

[19] S. M. Rump: INTLAB – INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Springer Netherlands, 1999.

[20] S.M. Rump. Rigorous and portable standard functions. *BIT*, 41(3):540–562, 2001.

[21] S.M. Rump. Inversion of extremely ill-conditioned matrices in floating-point. *Japan J. Indust. Appl. Math. (JJIAM)*, 26:249–277, 2009.

[22] S. M. Rump: Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[23] S.M. Rump. Fast interval matrix multiplication. *Num. Alg.*, 61(1):1–34, 2012.

[24] S.M. Rump and Kashiwagi M. Implementation and improvements of affine arithmetic. *Nonlinear Theory and Its Applications, IEICE*, 2(3):1101–1119, 2014.

[25] S.M. Rump. Mathematically rigorous global optimization in floating-point arithmetic. *Optimization Methods and Software*, 33(4-6):771–798, 2018.

[26] S.M. Rump. Verified error bounds for all eigenvalues and eigenvectors of a matrix. *SIAM J. Matrix Anal. Appl. SIMAX*, 43(4):1736–1754, 2022.

[27] S.M. Rump and T. Ogita. Verified error bounds for matrix decompositions. *SIMAX*, 45(4):2155–2183, 2024.

[28] S.M. Rump. Verified error bounds for sparse systems Part I/II. to appear in SIMAX, 2026.

[29] R. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comp.*, 35(151):817–832, 1980.

[30] T. Sunaga. Geometry of Numerals. Master's thesis, Univ. of Tokyo, Feb 1956.

[31] J. Titi and J. Garloff. Matrix methods for the tensorial Bernstein form. *Applied Mathematics and Computation*, (346):254–271, 2019.

[32] L. Trefethen. The SIAM 100-Dollar, 100-Digit Challenge. SIAM-News 35, 2002.

[33] S.J. Wright. A collection of problems for which Gaussian elimination with partial pivoting is unstable. *SIAM J. Sci. Comput. (SISC)*, 14(1):231–238, 1993.